Einführung in C

aus:

Happy Computer -- PASCAL / FORTH / C
Sonderheft 5/1986

Markt & Technik

Programmiersprachen – Babel läßt grüßen

Wer den Computer zu seinem Hobby oder Beruf macht, steht in mehrerer Hinsicht vor einer schwierigen Wahl. Er muß sich zwischen einer Vielzahl von Peripherie, Betriebssystemen und Programmiersprachen entscheiden. Hier zeigen wir Ihnen die wichtigsten Programmiersprachen, ihre Entwicklungsgeschichte und was sie können.

Kommunikation braucht Sprache. Wenn sich zwei Menschen miteinander unterhalten, benutzen sie dazu das gesprochene Wort oder, sofern sie nicht die gleiche Sprache verstehen, Hände und Füße. Dieses Prinzip läßt sich nicht ohne weiteres auf die Verständigung zwischen Mensch und Computer anwenden. Kommunikation ist der Datenaustausch zwischen mehreren Parteien, die einander verstehen müssen, jedoch ist ein Computer im Urzustand alles andere als verständig. Er besitzt lediglich eine mehr oder weniger große Anzahl von Fähigkeiten, die er sehr schnell ausführen, aber nicht selbständig koordinieren kann.

Befehl und Gehorsam

Programmiersprachen sind, im Gegensatz zur weitverbreiteten Meinung, kein Mittel, um sich mit dem Computer zu unterhalten. Sie dienen lediglich dazu, dem Computer über eine Kette von Befehlen mitzuteilen, was er Schritt für Schritt zu tun hat. Programmierung wurde lange Zeit unter dem Hauptaugenmerk der Mensch-Maschine-Kommunikation betrachtet. Der wichtigste Aspekt lag darin, zu Problemlösungen unter möglichst effizienter Nutzung der Maschinenkapazität zu gelangen. Seit Computer in jüngster Zeit eine stärkere Verbreitung erfahren haben, rückte jedoch ein zweiter Gesichtspunkt immer mehr in den Vordergrund: Die Programme werden komplexer und der Wartungsaufwand (Korrektur oder Erweiterung eines Programms) immer größer. Dadurch, daß an vielen Programmen große Teams über lange Zeiträume hinweg arbeiten, geraten Programmiersprachen auch mehr und mehr zum Kommunikationsmittel zwischen diesen Gruppen von Menschen. Das bedeutet, daß ein guter Programmierer immer seine Programme auch für seine Nachwelt verständlich gestalten, und seine Kunstfertigkeit nicht mit der Anwendung von Programmiertricks unter Beweis stellen sollte. Einem potentiellen Benutzer, der das Programm lesen und eventuell ändern muß, sollte die Funktionsweise möglichst schon beim Lesen klar werden.

Statt »Programmiersprache« wäre die Bezeichnung »Kommandosequenz« eigentlich richtiger. Was dem Computer befohlen wird, führt er geduldig und beliebig oft aus, einzige Bedingung ist ein fehlerfreies Programm. Einige Psychologen behaupten denn auch, die Beliebtheit von Computern sei darauf zurückzuführen, daß viele Menschen ihre diktatorischen Triebe beim Programmieren ausleben!

Wenden wir uns der Frage zu, die wohl jeden Computeranwender irgendwann einmal bewegt, nämlich, was denn welche Programmiersprache wohl zu leisten vermag.

Im Laufe der Computergeschichte, die seit den ersten Relaisrechnern Konrad Zuses nicht mehr als ein halbes Jahrhundert zählt, wurde eine Unzahl Programmiersprachen, Spracherweiterungen und Dialekte entwickelt. Dabei standen immer zwei Überlegungen im Vordergrund. Zum einen mußten die Hardwarevoraussetzungen berücksichtigt werden. Zum anderen orientieren sich die Programmentwickler an den zu bearbeitenden Problemstellungen und den Bedürfnissen der Anwender. Je nach Computertyp und Problemstellung werden vom Programmierer verschiedene Programmstrukturen (Module, Blockkonzept, Verbundtypen), spezialisierte Befehle und unterschiedliche Datentypen (Integer, Real, Complex, String, etc.) benötigt. Auf dieser Grundlage entwickelten sich neben den bekannten Sprachen eine unüberschaubare Anzahl Exoten, die meistens nur in einzelnen Universitäten eingesetzt wurden.

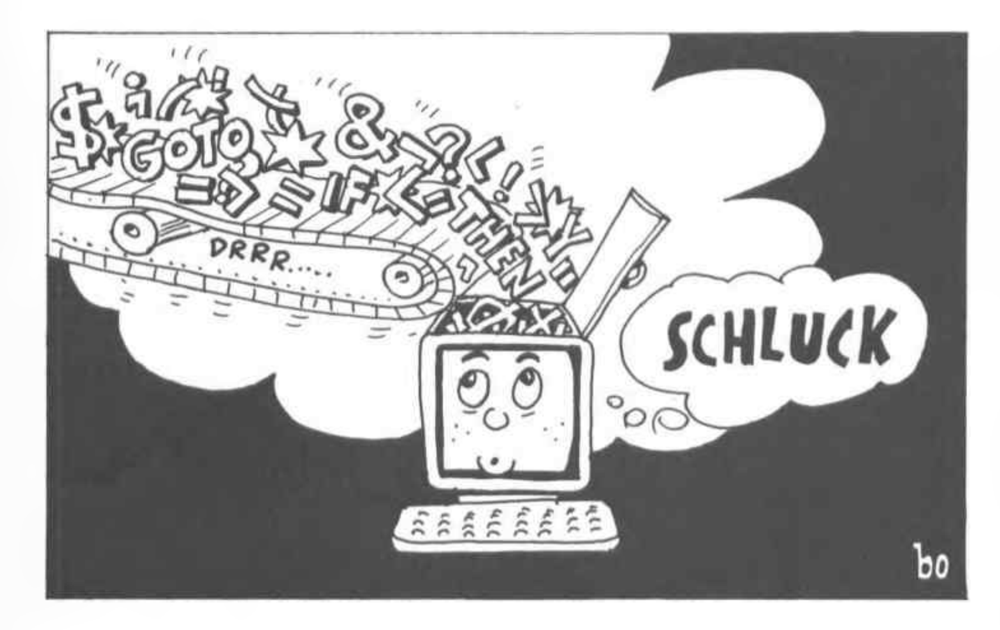
Vielfalt – Freud oder Leid?

An kaum einem Punkt scheiden sich die Geister in der Computerszene so stark wie in der Auswahl der Beurteilung der Programmiersprachen. Wer den Heimcomputer zu seinen Hobbies zählt, lernt in aller Regel zunächst fleißig Basic. Schließlich gehört ja der Basic-Interpreter zum Lieferumfang. Im Laufe der Zeit werden die eigenen Programme immer länger und unübersichtlicher, die einzelnen Programmteile sind durch ein unentwirrbares Geflecht von GOTO-Anweisungen miteinander verknotet (böse Zungen sprechen deshalb von »Spaghetti-Code«).

Mit dem wachsenden Bedürfnis nach Strukturierung, die in Basic nur jemand mit viel Selbstdisziplin erreicht, und nach Geschwindigkeit, die für Basic ein Fremdwort ist, sieht man sich nach Auswegen um. Dabei fühlt sich die eine Gruppe wie magisch von dem Begriff Assembler angezogen und begibt sich auf die unterste Sprachebene, um fortan in mühseliger Kleinarbeit Byte für Byte zu programmieren. Hiermit wird zwar ein Höchstmaß an Geschwindigkeit möglich, aber die Übersichtlichkeit kommt nach wie vor zu kurz. Die zweite Gruppe der Programmier-Gemeinde wendet sich deshalb modernen Hochsprachen zu, wie Pascal, Forth, Modula, Comal, Ada, C und so weiter. Diese Sprachen zwingen den Programmierer dazu, seine Programme modular (dies bedeutet, einzelne Aufgaben werden in »Paketen« oder »Modulen« zusammengefaßt) zu gestalten. Durch die so erreichte Übersichtlichkeit lassen sich Programme später von jedermann, Sprachkenntnisse vorausgesetzt, nachvollziehen und ändern. Zudem

sind einige dieser Sprachen sehr assemblernah, wie zum Beispiel Forth, womit auch Geschwindigkeit kein Problem mehr ist. Moderne Programmiersprachen unterstützen also Programmstrukturen und gehen ebenso mit Daten- und Kontrollstrukturen problemgerecht um.

Dennoch werden im professionellen Bereich (Universitäten, Verwaltung) »klassische« Sprachen bevorzugt, wie PL/1, Cobol und Fortran. Es sprechen auch gute Gründe dafür: So sind die neueren Programmiersprachen oftmals auf der vorhandenen Hardware noch nicht verfügbar, oder sie vertragen sich mit bereits vorhandenen Softwarekomponenten nicht. Ein anderer Faktor sind die Vorkenntnisse des Wartungspersonals und und und .... Jeder Informatikstudent, jeder Praktiker kann diese Reihe beliebig fortsetzen. Und schließlich ist gute Software immer noch der Triumph des Programmierers, nicht der der Programmiersprache.



Seit das Betriebssystem CP/M auf Computern wie dem Commodore 128, Schneider CPC, oder dem Atari ST einen neuen Frühling erlebt, werden diese Klassiker neuerdings auch auf Heimcomputern interessant. Sehr wahrscheinlich wird sich ein breiter Anwenderkreis hierfür finden. Welcher Programmierer begrüßt es nicht, wenn er seine Produkte teilweise in der »guten Stube« austesten kann?

Eine Sonderstellung unter den Programmiersprachen nehmen die Sprachen für »künstliche Intelligenz« (KI) ein. Deren bekannteste Vertreter heißen Lisp und Prolog. KI-Sprachen bauen auf einem grundlegend neuen Konzept auf, bei dem der Programmierer dem Computer im Dialog sein Problem mitteilt. Wegen des hohen Speicherbedarfs sind KI-Programme auf Microcomputern nur sehr eingeschränkt einsatzfähig. Dies wird sich jedoch bald ändern. Man darf gespannt sein, wie sich KI auf den 16-Bit-Computern entwickeln wird, die ja nicht nur in punkto Schnelligkeit, sondern auch im Speicherangebot einen ganz neuen Standard setzen. Der Künstlichen Intelligenz ist in diesem Sonderheft ein eigener Beitrag gewidmet.

Fassen wir zusammen: Sinnvoll lassen sich die Programmiersprachen in vier Gruppen unterteilen:

1. Assemblersprachen: Sie bieten den Vorteil, daß sie die Möglichkeiten der Hardware optimal ausnutzen. Assembler versteht jeder Prozessor unmittelbar. Jede höhere Programmiersprache muß beim Programmablauf erst in Assemblercode übersetzt werden und ist deshalb weniger universell. Gegen Assembler sprechen die mühselige Programmierung, die Gebundenheit an Prozessor und Hardware sowie die schwierige Wartung.

2. Klassische Hochsprachen: Sie sind zur Zeit am weitesten verbreitet, aber genügen wegen ihrer altertümlichen Konzeption den Anforderungen moderner Programmierung nicht mehr.

3. Moderne Hochsprachen: Sie können sich gegen die »Alteingesessenen« nur langsam durchsetzen. Sie bieten Strukturierungshilfen, ausgeprägte Möglichkeiten der Datenbeschreibung, und unterstützen die Selbstdokumentation des Programmtextes durch eine angemessene Verbalisierung.

4. KI-Sprachen sind heutzutage noch Gegenstand intensiver Forschungen und stellen eine völlig neue und andersartige Klasse von Programmiersprachen dar. In diesem Beitrag wollen wir uns deshalb nicht weiter mit ihnen beschäftigen.

Wie Sie gesehen haben, ist der Sprachenwald immer noch sehr dicht, auch wenn man sich auf die Auswahl der wichtigsten Programmiersprachen beschränkt. Ein bedeutendes Auswahlkriterium sind die qualitativen Merkmale. Bevor wir diese besprechen, noch zu einigen zentralen Begriffen:

Algorithmus – Jedem Programm liegen ein oder mehrere Lösungsverfahren zugrunde, oft auch als Algorithmus bezeichnet. Ein solcher Algorithmus ist definiert als eindeutige und vollständige Vorschrift zur Lösung einer Problemklasse mit einer Abfolge von Schritten, die in einem endlichen Zeitraum ausgeführt werden.

Daten heißen dabei die Objekte, die der Algorithmus bearbeitet.

Programm nennt man folgerichtig die maschinengerechte Aufbereitung der Daten und des Algorithmus. Diese Aufbereitung geschieht mit Hilfe einer künstlichen Sprache, eben der **Programmiersprache**. Diese setzt sich aus einer Menge von Zeichen zusammen, die ihrerseits nach bestimmten Regeln zusammengesetzt werden können. Die somit geschaffenen Sprachelemente werden von der Maschine unmittelbar (als Maschinensprache) oder unter Zuhilfenahme von Übersetzungsprogrammen (als Hochsprache) »verstanden«. Der Übersetzer stellt nichts weiter dar, als einen speziellen Algorithmus, der in der Lage ist, alle Befehle einer Hochsprache in den entsprechenden Assemblercode zu transformieren. Er ist grob vergleichbar mit einer Bibliothek von kleinen Assembler-Unterprogrammen, wobei jedem Befehl der Hochsprache eines dieser Unterprogramme zugeordnet ist.

Compiler und Interpreter

Übersetzungsprogramme gliedern sich in zwei Typen, die die gleiche Aufgabe auf unterschiedliche Weise erfüllen. Als erstes sind die **Compiler** zu nennen. Sie tauchten auch in der geschichtlichen Entwicklung zuerst auf. Der Compiler übersetzt den Programmtext (Quellcode) in einem oder mehreren Durchgängen (Passes) komplett in Assemblercode (Objektcode). Der Compiler selbst wird daher beim Programmlauf nicht mehr benötigt. Natürlich benutzt man für unterschiedliche Computer, mit verschiedener Hardware und Maschinensprache auch unterschiedliche Compiler.

Der zweite im Bund der Dolmetscher nennt sich **Interpreter**. Er ist der fleißigere von beiden. Zunächst muß der Quellcode direkt im Arbeitsspeicher des Computers abgelegt werden. Sodann beginnt das Interpretieren. Das heißt nichts weniger, als daß der Interpreter während des Programmlaufs Befehl für Befehl holen muß. Natürlich ist diese Vorgehensweise in höchstem Maße unökonomisch und langsam. Während ein Compiler die ersten drei der genannten vier Arbeiten nur genau einmal ausführen muß, beschäftigen sie den Interpreter bei jeder Programmwiederholung aufs neue.

Zu Beginn des Computerzeitalters, als Rechenzeit noch sehr teuer war, wurden daher ausschließlich Compiler entwickelt. Interpreter konnten sich erst mit höheren Prozessorleistungen durchsetzen. Sie werden auch heute noch in Profikreisen wegen ihres gemächlichen Arbeitstempos verschmäht.

Die Qualität von Programmiersprachen

Um beurteilen zu können, welche Eigenschaften gute Programmiersprachen charakterisieren, wenden wir uns zunächst der Frage zu, welche Anforderungen an ein hochwertiges Programm zu stellen sind.

Die Forderung nach Fehlerfreiheit erscheint auf den ersten Blick banal. Die Erfahrung zeigt nämlich, daß 100%ig korrekte Programmsysteme ab einem gewissen Umfang kaum noch möglich sind. Hat ein Programm eine gewisse Komplexität erreicht, ist es fast unmöglich, seine Korrektheit zu beweisen, also Testdurchläufe zu finden, die tatsächlich alle Eventualitäten berücksichtigen. Als Maß für die Korrektheit eines Programmes wird deshalb häufig von der Zuverlässigkeit gesprochen. Diese gibt die Wahrscheinlichkeit an, mit der ein Programm für eine Zahl von Anwendungsfällen in einer bestimmten Zeitspanne fehlerfrei arbeitet. Gerade die jüngste Vergangenheit hat hierfür im Bereich der Mikrocomputer einige negative Beispiele geliefert. So konnten Fehler in einigen Betriebssystemen oftmals erst nach der Auslieferung der neuen Geräte beseitigt werden.

Verständlichkeit deckt sich mit Forderungen nach Lesbarkeit, Überschaubarkeit und Selbstdokumentation. Programme sollten sich, sobald sie eine gewisse Länge überschreiten, in funktionelle Einheiten gliedern. Andernfalls wird der Programmierer oft, noch während er an ein und demselben Programm arbeitet, an der selbst produzierten Unordnung scheitern. Man differenziert unter dem Aspekt der Verständlichkeit zwischen der statischen und der dynamischen Programmstruktur. Die statische Strukturgestaltung dient dem Ziel, ein übersichtliches Layout des Programmtextes zu gestalten. Hiermit wird der menschlichen Wahrnehmung Rechnung getragen, die sehr stark auf optischen Wahrnehmungen beruht. Ergänzend trägt die dynamische Strukturierung dazu bei, daß Programmabläufe unmittelbar aus dem Quelltext ersichtlich werden.

Ein Faktor der an diese Zusammenhänge anknüpft, ist die Änderbarkeit von Programmen. Wartungsfreundlichkeit bedeutet einerseits, daß Programme leicht an modifizierte Aufgabenstellungen angepaßt werden können. Andererseits spielt die Portabilität eine Rolle, wenn ein Programm zum Beispiel in einer neuen Softwareumgebung lauffähig gemacht werden soll.

Universalität sollte es einem guten Programm ebenfalls ermöglichen, ähnliche Aufgabenstellungen und Abwandlungen zu lösen.

Im Zusammenhang mit der Benutzerfreundlichkeit sollten Programme einen Dialog mit dem Benutzer ermöglichen und Eingabefehler abfangen, ohne falsche Ergebnisse oder gar Systemabstürze zu liefern.

Effizienz hat im Laufe der Entwicklung stark an Bedeutung verloren. Zu Zeiten, da Speicherplatz noch Mangelware war, galten die kürzesten Programme als die besten. Das ging natürlich zu Lasten der Übersichtlichkeit. Effizienzstreben wird heute daher nur noch mit Skepsis betrachtet.

Die hier genannten Qualitätsmerkmale stehen offensichtlich in positiver und negativer Wechselwirkung zueinander. So geht Übersichtlichkeit meistens zu Lasten der Effizienz, verbessert dagegen aber die Änderbarkeit.

Beginnen wir jetzt damit, die Anforderungen an eine ideale Programmiersprache zu formulieren. Die Qualität einer Computersprache läßt sich danach beurteilen, inwieweit sie die Entwicklung von Programmen unterstützt, die den uns bekannten Anforderungen entsprechen. Die folgenden skizzierten Merkmale müssen im engen Zusammenhang miteinander betrachtet werden.

Beginnen wir mit einem Punkt, der besonders den Einsteiger interessieren wird:

Die Erlernbarkeit einer Sprache hängt wesentlich von deren Struktur und Umfang ab. Sie ist sehr viel einprägsamer, wenn die Zahl der Schlüsselwörter gering und das Sprachkonzept durchgängig ist. Ebenso gewährleistet eine einfache Benutzung, wenn der Programmierer sein Problem bequem mit einer breiten Palette von Ausdrucksmöglichkeiten lösen kann.

Die Einheitlichkeit ist ein ebenso wichtiger wie unscharfer Begriff. Er soll im großen und ganzen bedeuten, daß für eine bestimmte Leistung möglichst nur genau ein Sprachmittel zur Verfügung steht.

Kriterien für eine ideale Sprache

Kompaktheit steht mit Einfachheit in Einklang. Hierbei ist nicht von der »Würze der Kürze« die Rede, sondern vielmehr von der Mächtigkeit der Sprachkonzepte. Der Bauplan der Sprache muß eine geringe Anzahl verschiedener Grundkonzeptionen aufweisen, wie mathematische Operationen, Ein- und Ausgabe, Datenstrukturierung. Andererseits soll Kompaktheit ein gewisses Maß an Redundanz (= alles was man in der gleichen Sprache auf andere Weise auch darstellen kann) an der richtigen Stelle nicht verhindern. Dies fördert die Verständlichkeit und Zuverlässigkeit. Beispielsweise sind Datentypen im Grunde genommen redundant, doch wer möchte sich schon mit einer Sprache herumschlagen, die ausschließlich den Datentyp »Zeichen« kennt?

Die Sprache Basic wird von vielen als katastrophal eingestuft. Das hat einen guten Grund: Die Forderung nach Lokalität wird von Basic so gut wie gar nicht unterstützt. Mit diesem Begriff ist gemeint, daß die Teile einer bearbeiteten Aufgabe, die logisch zusammengehören, auch im Programmtext in physischer Nachbarschaft stehen sollten. Die berühmt-berüchtigten Sprungbefehle bewirken jedoch das genaue Gegenteil.

Ein ganz anderes Kriterium ist die Sicherheit der Programmiersprache. Dazu zählt das Unterstützen der Fehlerfreiheit eines Programmes durch Sprachelemente. »On Error Goto« ist ein solcher weitverbreiteter Befehl. Des weiteren sind Sprachelemente zu nennen, die die Testphase des Programms fördern.

Der »Wildwuchs« der Implementierungen (= Anpassungen eines bestimmten Computers) bei Programmiersprachen, insbesondere bei den älteren, ist hinlänglich bekannt. Dialekte, Erweiterungen, aber auch Einschränkungen beeinträchtigen die Portabilität von Programmen im besonderen Maße. Mit der Standardisierung

Fortsetzung auf Seite 10

befassen sich daher nationale und internationale Institute. Die wichtigsten sind das Deutsche Institut für Normung (DIN), das American National Standards Institute (ANSI) und die International Organization for Standardization (ISO). Die Benutzer akzeptieren allerdings die Standards unterschiedlich. So sind Standards von Basic nahezu unbekannt, während sie sich bei Sprachen wie Pascal oder Fortran zunehmend durchsetzen.

Was für die Effizienz bei Programmen gesagt wurde, verkehrt sich bei den Sprachen in das genaue Gegenteil: Für den Übersetzer ist sie von großer Wichtigkeit. Zum einen sollte der Übersetzungsvorgang effizient sein, da bei der Fertigstellung eines Programmes die Zahl der Testläufe meistens sehr hoch ist. Der wichtigere Grund aber ist, daß das erzeugte Maschinenprogramm im Hinblick auf Rechenzeit und Speicherbedarf optimiert werden sollte. Sogenannte »Optimierende Übersetzer« sind teilweise in der Lage, das erzeugte Maschinenprogramm effizienter zu gestalten, als dies der Benutzer durch Programmiertricks erreichen kann.

Von Zuse bis Ada

Wir besitzen nun ein gutes Handwerkszeug, um Programmiersprachen nach den wichtigsten Gesichtspunkten theoretisch zu beurteilen. Fassen wir zusammen: Die entscheidenden Anforderungen an eine Sprache heißen Strukturierungshilfen, Selbstdokumentation, Datenbeschreibung, Benutzerfreundlichkeit und Zuverlässigkeit. Doch was nützt all die graue Theorie, wenn wir die Sprachen nicht kennen? Im folgenden werden daher die wichtigsten unter den bisher behandelten Aspekten und im Rahmen ihrer geschichtlichen Entwicklung vorgestellt.

Die Entstehung der Programmiersprachen orientiert sich immer auch an den Voraussetzungen der Hardware. Dies gilt insbesondere auch für die Gründerjahre.

Als geistiger Urvater der Rechenmaschinen mit Programmsteuerung darf der Engländer Charles Babbage gelten. Er begann 1833 mit der Konstruktion digitaler Rechenautomaten. Er legte seinen Maschinen aus Zahnrädern, Kurbeln und Hebeln zwei wichtige Erfindungen zugrunde, nämlich die Lochkartensteuerung und das Prinzip eines dekadischen Zählrades mit automatischem Zehnerübertrag. Babbages Projekte waren aber wegen fertigungstechnischer Schwierigkeiten nur in der Theorie funktionsfähig. Erst Elektromechanik und später die Elektronik machten die Rechenautomaten langsam zu Computern.

Der erste Rechenautomat der Welt mit Programmsteuerung wurde 1941 von Konrad Zuse in Betrieb genommen. Die Zuse Z3 war ein Relaisrechner, der bereits mit Dualzahlen arbeitete und zur Darstellung Gleitkommazahlen benutzte. Mit der Z3 waren neben den vier Grundrechenarten auch das Ziehen von Quadratwurzeln und das Potenzieren möglich. Ein Nachbau des historischen Modells, das im Krieg zerstört wurde, steht im Deutschen Museum in München. Der erste programmierbare Rechner Amerikas entstand 1944. Er wurde von dem Mathematiker Howard H. Aiken mit Unterstützung von IBM entwickelt und auf den Namen »Mark I« getauft. Er war jedoch ein Ungetüm von 16 m Länge und 35 Tonnen Gewicht und zudem langsamer als die früher entwickelte Z3.

Der Phase der Relaisrechner setzte der Einsatz von Elektronenröhren rasch ein Ende. Der bekannte ENIAC war die erste vollelektronische Rechenanlage der Welt und wurde 1945 in den USA fertiggestellt. Er erreichte gegenüber den Relaisrechnern bereits die 2000fache Rechengeschwindigkeit.

Während die Lochkarte nur eine starre Programmsteuerung ermöglichte (keine Schleifen, keine logischen Entscheidungen) begann man bald, sich über die flexible Speicherprogrammierung Gedanken zu machen. Als erstem gelang es dem Amerikaner John Neumann das genannte Problem auf einem Rechner zu verwirklichen. Der bereits 1944 von Neumann konzipierte Computer (EDVAC) erfüllte folgende Forderungen: Das Programm mußte, wie auch die zu verarbeitenden Daten, in der Maschine gespeichert werden. Außerdem benötigte man bedingte Befehle wie Vorwärts- und Rückwärtsverzweigungen. Jeden Befehl konnte zudem die Maschine selbst, wie jeden anderen Operanden ändern.

Befehle bestanden aus einem Operations- und einem Adreßteil. Im Operationsteil wird eine Angabe gemacht was zu tun ist (zum Beispiel Ausführung einer Multiplikation), der Adreßteil zeigt an, wo sich die zu verarbeitenden Daten befinden und wohin sie anschließend zu übertragen sind. Hier ist also die Rede von den ersten Assemblersprachen, an deren Grundprinzipien sich bis heute nicht geändert hat.

Der Schritt von der starren Programmsteuerung zum flexiblen Programm leitete die Wende vom Rechner zur Datenverarbeitung ein. Die Röhrencomputer der ersten Generation erreichten mit dem SSEC (Selective Sequenz Electronic Calculator) Ende der vierziger Jahre einen Höhepunkt. Dieser besaß nicht weniger als 12000 Elektronenröhren und etwa 21500 Relais und wurde von 36 Lochstreifenlesern gesteuert. Er führte die Berechnungen der Mondbahn durch, die 20 Jahre später im Apollo-Raumfahrtprogramm verwertet wurden. Mit dem Einzug der Transistortechnik und später mit den integrierten Schaltkreisen wuchsen fortan Rechnerleistung und Speicherkapazität immer schneller. Dies war die Voraussetzung für die Schaffung der höheren Programmiersprachen.

Die frühen Jahre – Fortran

Fortran ist die älteste der hier behandelten Hochsprachen und setzt einen Meilenstein in der Geschichte. Anfang der fünfziger Jahre wuchs die Zahl der Computer rasch. An Serienfertigung war noch nicht zu denken und es war jedes Gerät ein Einzelstück mit eigener Hardware und eigenem Assembler. So wurde bald der Wunsch nach einer Programmiersprache laut, die übertragbar und einfach zu programmieren sein sollte. 1952 wurde der Grundstein für Fortran gelegt, zu einer Zeit, da die Programmierung nur wenigen Spezialisten und ausschließlich in Assembler möglich war. John W. Backus war einer der Federführenden, dem die Programmiergemeinde Fortran zu verdanken hat.

Der Hauptgrund für die Entwicklung war die Schwerfälligkeit der Assemblerprogrammierung. 75 Prozent der Kosten eines Rechenzentrums verursachte damals die Fehlersuche. Verständlichkeit war daher ein wesentliches Entwurfsziel. Dadurch, daß die teuere Hardware optimal ausgenutzt werden mußte, waren die Rahmenbedingungen für Fortran bereits vorgezeichnet. Vorrangig wurden Sprachelemente implementiert, die der Speicher- und Laufzeiteffizienz nachkamen. Einige dieser Konzepte werden noch heute als sehr nachteilig angesehen – sind aber immer noch in Fortran enthalten.

1955 erschien ein Programmierhandbuch, und zwei Jahre später wurde die erste Implementierung auf einer IBM 704 freigegeben. Damit stand Fortran erstmals einer breiten Zahl von Programmierern zur Verfügung.

Der Name steht für FORmula TRANslating system (Formelübersetzer). Und genau dort liegt auch der Anwendungsschwerpunkt der Sprache. Rechnerische Probleme lassen sich in ihr leicht und natürlich ausdrücken. Damit wird

der Erlernbarkeit der Sprache Rechnung getragen. Im ingenieurwissenschaftlichen und mathematischen Bereich gilt Fortran auch heute noch als die wichtigste Programmiersprache. So bietet sie zum Beispiel neben den allgemein gebräuchlichen Zahlentypen Real (Fließkommazahlen) und Integer (Ganze Zahlen) auch noch den Typ »Double Precision« für Rechnungen mit höherer Genauigkeit sowie »Logical« für boolsche Operationen. Großrechner-Versionen beinhalten zudem noch den Typ »Complex«, der in der theoretischen Elektrotechnik eine sehr wichtige Rolle spielt.

Das Format dieser Sprache ist streng zeilenorientiert und erlaubt normalerweise nur einen Befehl je Zeile. Das hängt damit zusammen, daß Fortran zunächst als lochkartenorientierte Sprache entstand. Grundsätzlich mußte man damals für jede neue Anweisung eine neue Lochkarte (beziehungsweise Zeile) verwenden. Wie auch in Basic, das später aus Fortran entstand, mußte bei den ersten Versionen viel mit dem Goto-Befehl umhergesprungen werden. Neuere Versionen wie Fortran V und Fortran 77 bieten demgegenüber schon strukturierende Sprachelemente wie IF....THEN....ELSE...ENDIF.

Nachdem die 1958 geschaffene Version Fortran II eine mäßige Verbreitung gefunden hatte, entstand 1962 das in weiten Kreisen akzeptierte Fortran IV. Den fortschreitenden Auswüchsen immer neuer Versionen wurde 1966 Einhalt geboten, mit einer Version, die größtenteils mit Fortran IV identisch war. Schließlich überarbeitete das ANSI Fortran 66 im Jahr 1977 nochmals und beseitigte einige eklatante Mängel.

Unter CP/M ist Fortran derzeit für fast alle Mikrocomputer mit Z80-Prozessor erhältlich, ebenso wie eine Reihe von Fortran-Implementationen für MS-DOS-Computer.

Cobol – die Geschäftige

Die Programmiersprache Cobol entstand 1959 auf Initiative des US-Verteidigungsministeriums. Zu dieser Zeit begann Fortran sich gerade auszubreiten. Was noch fehlte, war eine Sprache für den kommerziellen und kaufmännischen Einsatz. So entwickelte man Cobol mit dem Ziel, große Datenbestände verarbeiten zu können und die Ein/-Ausgabe zu unterstützen. Insbesondere die ersten Fortran-Versionen waren hierfür ungeeignet. Ende der fünfziger Jahre wurde die Codasyl-Entwicklungsgruppe aus Vertretern der Computerindustrie und der amerikanischen Regierung gegründet. Schon 1960 stellte diese Gruppe die erste Version mit der Bezeichnung Cobol-60 vor. Sie war wesentlich an die weniger bekannte Sprache Comtran (COMmercial TRANslator) angelehnt. Aufgrund der hastigen Entwicklung von Cobol innerhalb eines halben Jahres ergaben sich viele Ungereimtheiten, die sich teilweise durch alle Neuentwürfe hindurchschleppten und auch heute noch nicht ganz beseitigt sind. Cobol-61 war dann die Grundlage für alle späteren Versionen. Sie war zu Cobol-60 nicht kompatibel. 1965 wurde als wesentliche Neuerung die Unterstützung von Massenspeichern und Tabellen mit eingebracht.

Die Sprachelemente von Cobol sind je nach ihrer Funktion in Module zusammengefaßt. Das ANSI entwickelte bis 1974 einen zwölf Module umfassenden Standard, namentlich Cobol ANS-74. Er wurde 1980 nochmals verbessert. Den jeweils neuesten Stand veröffentlicht das CODASYL-Komitee im Abstand von drei Jahren. Der Standard für die Bundesrepublik Deutschland ist in der DIN-Norm 66028 nachzulesen.

Die kurze Darstellung der Entwicklungsgeschichte läßt erkennen, daß Cobol ebenfalls eine alte Sprache ist. Cobol-Programme müssen aus heutiger Sicht als mangelhaft angesehen werden.

Ursprünglich verfolgte man wie bei keiner anderen Sprache das Ziel der Lesbarkeit des Programmtextes. Die Sprache sollte dann auch leicht erlernbar sein. Der Programmtext erinnert stark an (englische) Prosa. Cobol-Programme simulieren nämlich die natürliche englische Sprache. So wird zum Beispiel jeder Befehl mit einem Verb eingeleitet. Die Grundrechenarten stehen nicht als Symbole, sondern als Befehlswörter (add, divide) zur Verfügung. Ebenso werden logische Operatoren ausgeschrieben. Ein Beispiel: Wenn die Variable A größer ist als Null, soll der Variablen B die Summe der Variablen C und A zugeordnet werden. In Basic würde man das so formulieren:

```
IF A>0 THEN B= C+A
```

Daraus wird in Cobol:

```
IF A GREATER THAN ZERO ADD C TO
A GIVING B.
```

Daß so aus komplizierteren mathematischen Formeln monströse Gebilde werden, leuchtet ein. Da höhere mathematische Funktionen in Cobol ganz fehlen, ist die Sprache für wissenschaftliche Anwendungen völlig ungeeignet.

Derartige Beispiele verdeutlichen, daß das Entwicklungsziel von Cobol nicht sinnvoll erreicht wurde. Dennoch ist Cobol auch heute noch die weltweit am stärksten verbreitete Programmiersprache. Ganze Rechenzentren arbeiten mit ihr. Die hohen Investitionen und die Gewöhnung des Personals an diese Sprache wird auch in Zukunft für ihren Erhalt sorgen. Zudem erreicht auch noch keine neuere Programmiersprache die Cobol-Domäne Datenorganisation.

Wer einen Mikrocomputer mit den Betriebssystemen CP/M oder MS-DOS (PC-DOS) besitzt, kann in Cobol einsteigen.

PL/1 – von allem etwas

Fortran und Cobol sind charakteristisch für die strikte Trennung in kommerzielle und technisch-wissenschaftliche Anwendungen zu Beginn der sechziger Jahre. Daneben wurden Computer nur noch in Spezialgebieten eingesetzt. Im Laufe der Zeit traten aber die Merkmale der naturwissenschaftlich-technischen Bereiche in den betriebswirtschaftlichen Anwendungen immer mehr hervor und umgekehrt. So waren die kommerziellen Anwender mehr und mehr auf Methoden aus der Statistik, Operations Research und Ökonomie angewiesen. Mathematiker und Ingenieure stellten zunehmend höhere Ansprüche an Datenverwaltung und an die Ein-/Ausgabeunterstützung.

Als logische Konsequenz kamen die Anbieter den neuen Bedürfnissen mit einer universellen Hard- und Softwarekonfiguration nach. Hardwareseitig entwickelte IBM die Rechnerfamilie /360 mit dem Betriebssystem OS/360. Diese Anlagen zählten sich bereits zur dritten Computergeneration (das heißt, die Schaltkreise waren in Hybridtechnik ausgelegt, einer unmittelbaren Vorstufe der integrierten Schaltkreise).

Bei den Überlegungen zu einer neuen Sprache war man zunächst von Fortran ausgegangen. Die Organisation SHARE (Society for Help to Avert Redundant Effort), eine Vereinigung wissenschaftlicher IBM-Anwender, einigte sich mit der Firma IBM auf die Gründung eines Sprachkomitees. Zunächst war die Rede von Fortran VI. Man gelangte aber schon nach kurzer Zeit zu der Erkenntnis, daß die gewünschten Verbesserungen eine Kompatibilität mit Fortran unmöglich machten. Die Anlehnung an Fortran hätte außerdem die große Gruppe der kommerziellen Verwender abgeschreckt. So entschied man sich für die Entwicklung einer gänzlich neuen Sprache. Deren wichtigsten Entwurfsprinzipien waren: allgemeine Einsetzbarkeit und weitgehende Ausdrucksfreiheit. Der Sprachaufbau sollte modular sein und Testhilfen sowie Möglichkeiten zur

Fehlerbehandlung bieten. Ein großenteils gegenläufiges Ziel bestand in den Forderungen, den vollen Zugriff auf Hardware- und Betriebssystemleistungen bei gleichzeitiger Maschinenunabhängigkeit zu gewähren.

Nach vielen drastischen Überarbeitungen hatte sich der Sprachumfang bis 1965 stabilisiert. Nachdem die Abkürzung für NPL (New Programming Language) bereits vergeben war, einigte man sich schließlich auf PL/I (Programming Language one). Im August 1966 wurde dann der erste Compiler für eine IBM /360 freigegeben. Schließlich verabschiedeten das ANSI und die ECMA, ein europäisches Standardisierungskomitee, einen vorläufig endgültigen Standard. Die Verbreitung von PL/1 nahm zunächst rasch zu, wurde aber später den gesetzten Erwartungen nicht gerecht.

Kommen wir nun zu der Sprache, die jeder, und wenn nur vom Hörensagen, kennt. Sie kann sich rein zahlenmäßig im Mikrocomputerbereich als die am

In den Sprachumfang von PL/1 wurden viele Konzepte aus Fortran, Cobol, Algol und Jovial übernommen (letztere werden hier wegen ihrer geringen Verbreitung nicht behandelt). Leider gelang es nicht, sich bei der Auswahl nur auf die guten Eigenschaften der Vorgänger zu beschränken. Zudem ist der Sprachumfang riesig. Eine fast unüberschaubare Anzahl von Schlüsselwörtern zwingt zu Maßnahmen, durch die der Programmierer auch mit Teilmengen der Sprache sinnvoll arbeiten kann. Daher ergeben sich für die PL/1 -Syntax sehr freizügige Vorschriften. Einerseits existiert keine Zeilenstruktur, Zwischenräume, Einrücken und Kommentare dürfen fast beliebig verstreut werden. Da passiert es dann auch nicht selten, daß beim Programmieren das Format aus allen Fugen gerät. Außerdem sind die Schlüsselwörter nicht reserviert, wohl wegen ihrer großen Anzahl. Gebilde wie

```
IF ELSE=THEN THEN IF=ELSE; ELSE
IF=THEN
```

tragen wohl zur Verwirrung jedes hoffnungsvollen Programmierers bei. Ein wesentlich angenehmerer Fortschritt ist andererseits das ausgeprägte Blockkonzept der Sprache. PL/1 orientiert sich wesentlich an Prozeduren. Allgemein kann gesagt werden, daß die Einarbeitung in PL/1 viel Zeit braucht. Wer damit trotz aller Warnungen beginnen will, kann dies unter CP/M oder auf PC-Kompatiblen tun.

weitesten verbreitete Sprache der Welt rühmen. Sie ist schon fast als Teil der Allgemeinbildung zu betrachten. Ganz anders als bei den »großen professionellen« wurde die Entwicklung von Basic (Beginners All-purpose Symbolic



Instruction Code) nicht durch eine Industrie- oder Militärlobby getragen. Die Ausrichtung der Sprache kommt denn auch in ihrem Namen zum Ausdruck: Sie wendet sich an den Anfänger und soll für jeden Zweck geeignet sein.

Basic – Basis für Einsteiger

Ihre geschichtliche Entwicklung erklärt viele Aspekte des Sprachkonzepts. Basic wurde von Thomas Kurtz und John Kemeny von 1956 bis 1971 in den USA am Dartmouth College entwickelt. Ziel war es, Studenten, die sich nicht ausschließlich mit Ingenieurwissenschaften beschäftigten, das Programmieren zu erleichtern. So schlugen sich denn auch die Erfordernisse der allgemeinen Ausbildung einer Universität in der Sprache nieder: Bei der angesprochenen Zielgruppe erschien ein eigener Programmierkurs nicht erforderlich. Vielmehr sollte das Programmieren im Rahmen der Mathematikvorlesungen gelehrt werden. Hieraus erklärt sich auch die Ausrichtung von Basic auf mathematische Probleme. Die neue Sprache sollte leicht erlernbar und leicht zu benutzen sein. Dies war nach Meinung von Kurtz und Kemeny bei Fortran und Algol nicht der Fall. So erklärt sich auch, daß bei Basic nicht, wie bei PL/1, auf Bewährtes zurückgegriffen wurde.

Im Gegensatz zu den bisher behandelten Sprachen wurde Basic als vollständiges Programmiersystem konzipiert. Der Benutzer kann im Dialog mit Basic arbeiten, ohne die Basic-Umgebung zu verlassen. Hierzu existiert der »Direkt-Modus«, der das Editieren, Ausführen und Speichern von Programmen unterstützt. Basic ist zudem eine typische Interpreter-Sprache, für die allerdings auch verschiedene Compiler erhältlich sind.

Mit der Entwicklung des ersten Compilers begannen Kurtz, Kemny und eine Gruppe Studenten 1963. Im Mai 1964 wurde dann das erste Basic-Programm ausgeführt; die erste Version kannte nur 14 Instruktionen. Diese Minimalausstattung wurde am Dartmouth College bis 1971 in insgesamt sechs Versionen schrittweise vervollständigt. Seither nahmen die Autoren keine Veränderungen mehr vor. Das bedeutet natürlich nicht, daß Basic jemals eine wirkliche Standardisierung erfahren hätte. Im Gegenteil: Durch das Fehlen einer Interessenvertretung professioneller Anwender und durch die enorme Verbreitung der Sprache wucherten die Basic-Versionen fast uferlos. Für nahezu jeden Mini- und Mikrocomputer und selbst auf Taschenrechnern ist Basic erhältlich. Da von Herstellerseite nach dem Motto verfahren wurde »jedem Topf ein anderer Deckel« ist Kompatibilität ein Fremdwort. Auch die Bemühungen der ECMA und ANSI in den letzten Jahren waren kaum von Erfolg gekrönt. Lediglich ein »Minimal Basic« wurde kompromißbereit zum Standard deklariert. Diese Teilmenge ist der ersten Sprachbeschreibung von 1964 sehr ähnlich. Lediglich im Heimcomputerbereich gelang es mit den MSX-Computern erstmals, einen weitestgehenden Basic-Standard für Geräte verschiedener Hersteller zu schaffen. Haar in der Suppe war aber, daß die MSX-Computer wenig Verbreitung fanden.

Ein Basic-Programm besteht aus nummerierten Zeilen in aufsteigender Folge. Dabei sind die Zeilennummern aus einem festgelegten Intervall zu wählen. Jeder Zeilennummer folgen eine oder mehrere Instruktionen. Die

Zeilennummern legen die logische Reihenfolge der auszuführenden Anweisungen fest. Darüber hinaus dienen sie als Orientierung für die Sprungbefehle. Beim Editieren des Programmtextes lokalisieren sie die Zeilen. Basic ist noch stärker zeilenorientiert als Fortran. Das geht so weit, daß eine Basic-Anwendung durch die Zeilenlänge begrenzt wird. In einigen Basic-Versionen sind Fortsetzungszeilen mit dem Zeichen »&« möglich.

Als Datentypen sind in Basic nur numerische Daten und Zeichenketten vorgesehen. Eine Unterscheidung in ganzzahlig und Fließkomma, wie es von anderen Sprachen her bekannt war, wurde der Einfachheit wegen bewußt vermieden, ist aber dennoch auf vielen Computern implementiert. Für Datenstrukturen stehen nur ein- und mehrdimensionale Felder (Arrays) zur Verfügung. Deren Elemente können je nach Version entweder nur einzeln manipuliert werden, oder es stehen spezielle Matrizenoperationen bereit.

Die Sprunganweisungen sind, wie bereits erwähnt, vielen ein Greuel. Zwar bieten moderne Basic-Versionen viele Befehle, die das strukturierte Programmieren unterstützen, sie erfordern aber genaue Vorausplanung eines Programms. Im Regelfall endet bei längeren Programmen ein »Drauflosprogrammieren« im Chaos aus GOTO, GOSUB und FOR...NEXT.

Des weiteren beinhaltet Basic, je nach Ausstattung und Computer, Befehle für die Ein-/Ausgabe, mathematische Funktionen, Grafik, Tonerzeugung und den Zugriff auf das Betriebssystem. Hierauf im einzelnen einzugehen würde zu weit führen. Es sei daher auf entsprechende Fachliteratur verwiesen.

Pascal – strukturiert und einfach

Ein moderner Klassiker unter den Programmiersprachen ist Pascal. Der Name ist ausnahmsweise keine Abkürzung. Er wurde zu Ehren des französischen Mathematikers Blaise Pascal gewählt, der 1642 im Alter von 19 eine der ersten funktionsfähigen Rechenmaschinen konstruierte.

Die Wurzeln dieser Programmiersprache reichen in das Ende der sechziger Jahre zurück. Zu jener Zeit stellten Nikolaus Wirth und C.A.R. Hoare Überlegungen an, auf der Basis von Algol 60 eine Nachfolgesprache zu entwickeln. Algol 60 bot schon damals ein zufriedenstellendes Sprachkonzept. Das gilt besonders für die Strukturierung des Programmablaufs und des Programmtextes. Wegen dieser Vorteile wurde Algol zur Grundlage einer ganzen Klasse von Programmiersprachen, die bei Pascal beginnt und vorläufig von Ada gekrönt wird. Eine der Schwächen ist dagegen die unzureichende Datenstrukturierung. Ebenso wie Basic und Fortran kennt Algol 60 nur das Array. Die später folgende Version Algol 68 war ähnlich wie PL/1 zu umfangreich und unhandlich. Mit der Entwicklung von Pascal verfolgte man das entgegengesetzte Ziel. Der Schweizer Professor Nikolaus Wirth formulierte bei der Entwicklung der Sprache an der ETH Zürich die folgenden Schwerpunkte: Pascal sollte nur grundlegende Sprachkonzepte enthalten. Diese sollten natürlich definiert sein und das Erlernen des strukturierten Programmierens als eine systematische Disziplin unterstützen. Des weiteren sollte sie sich effizient auf allen Computern implementieren lassen.

Die erste vorläufige Version entstand 1968. Die vollständige Beschreibung eines Compilers und der Sprache selbst war 1971 fertig. Das 1974 erschienene Benutzerhandbuch »Pascal User Manual and Report« enthält eine Sprachdefinition, die heute als Wirth-Standard bezeichnet wird. Die verbreiteten Versionen Turbo- und UCSD-Pascal enthalten demgegenüber noch einige Erweiterungen, die vor allem Grafik und Zeichen-Strings betreffen.

Durch die leichte Erweiterbarkeit von Pascal entstanden bald viele Dialekte. Deshalb und wegen der weltweiten Anerkennung der Sprache nahmen sich verschiedene Normeninstitute diesem Problem an. Die ISO setzte schließlich 1980 einen Standard fest, der in der DIN-Norm 66256 nachzulesen ist. Strukturierung bedeutet nicht nur, daß das Programm übersichtlich ist, sondern daß sich der Vorgang des Programmierens in verschiedene Aktionen aufteilt. Bevor man sich an den Computer setzt, sollte man das Problem genau analysieren und in Aufgabenpakete zerlegen. Dann ist für jedes Paket ein Algorithmus zu bestimmen und in Pascal zu formulieren. Erst dann beginnt die Tipparbeit. Pascal-Programme entstehen also auf dem Papier und weniger am Bildschirm. Diese Vorgehensweise erreicht eine niedrige Fehlerquote und damit sinkt auch die Zahl der Übersetzerdurchläufe, was bei einer typischen Compilersprache wie Pascal sehr angenehm ist.

Ein Pascal-Programm ist klar gegliedert in einen Vereinbarungs- und einen Anweisungsteil. Vereinbart werden zuerst alle Variablen, Konstanten und deren Datentypen. Im Anweisungsteil werden die Aufgabenpakete in Procedures formuliert. Jede Procedure enthält einen eigenen Namen. Das eigentliche Hauptprogramm besteht dann nur noch aus dem Aufrufen der Procedures und steht im Programmtext ganz am Ende.

Daß der GOTO-Befehl in Pascal enthalten ist, verwundert eigentlich. Angesichts der Strukturbefehle IF....THEN-CASE kann man gut auf ihn verzichten. Der Vorrat an Datentypen eröffnet gegenüber Basic ganz neue Möglichkeiten. Man unterscheidet hier zwischen einfachen Typen, strukturierten Typen und Zeigertypen. Integer, Char, Boolean und Real zählen zu den einfachen Typen. Boolean bezeichnet eine Variable, der nur die Werte False oder True zugeordnet werden können. Array, Record, Set und File stehen als strukturierte Typen zur Verfügung. Set bezeichnet eine Menge. Auf diesen Typ sind die üblichen Mengenoperationen Vereinigung, Durchschnitt, Differenz, Untermenge und Elementüberprüfungen anwendbar. Record ermöglicht Verbundvariablen. Es lassen sich so unterschiedliche Variablentypen zu einer Variable zusammenfassen. Record war ursprünglich für kommerzielle Anwendungen gedacht (Tabellendarstellung). Demgegenüber werden mit dem Typ File nur Variablen eines einzigen Typs verkettet. Der Typ Zeiger (pointer) schließlich ermöglicht verkettete Listen und deren bequeme Manipulation, sowie Baumstrukturen. Wem das noch nicht reicht, der kann sich in Pascal weitere einfache Datentypen selbst definieren.

Pascal ist vielseitig und erzieht zum strukturierten Denken. Seine Verarbeitung, vor allem im akademischen Bereich, ist folgerichtig sehr hoch. So liegen denn auch für fast alle rechnereigenen Betriebssysteme wie auch für CP/M und MS-DOS Pascal-Versionen vor.

Forth – die etwas andere Sprache

Forth entstand Anfang der siebziger Jahre. Charles H. Moore entwickelte die Sprache ursprünglich zur Steuerung von Radioteleskopen. Er arbeitete dazu mit einem IBM-1130, einem Computer der dritten Generation. Das Endprodukt war aber so mächtig, daß es Moores Computer als einen der vierten Generation erscheinen ließ. Er wollte der neuen Sprache daher den Namen Fourth geben. Namen mit mehr als fünf Buchstaben waren auf dem IBM jedoch nicht erlaubt. So wurde das »u« ein Opfer dieser technischen Unzulänglichkeit.

Forth ist interaktiv wie Basic. Das heißt, es existiert sowohl ein Interpreter als auch ein Compiler. Programme können somit erst im Direktmodus »häppchenweise« getestet und anschließend compiliert werden. Des weiteren verbindet Forth Merkmale der Assemblersprachen mit denen der Hochsprachen.

Die Strukturierung in Forth entsteht durch die Definition immer neuer Worte. Der ohnehin schon große Sprachumfang nimmt beim Programmieren ständig zu. Beliebige Befehle können zu einem neuen Befehl zusammengefaßt werden, der dann sofort wieder in weitere Befehle mit eingebaut werden kann. Schließlich steht für das gesamte Programm ein einziger Befehl am Ende dieser Kette.

Selbstverständlich stellt Forth auch die Kontrollstrukturen zur Verfügung, die bereits für Pascal angesprochen wurden, wie IF.....ELSE....ENDIF, BEGIN...UNTIL, BEGIN...WHILE etc. Das berüchtigte GOTO fehlt hier ganz, ergäbe auch bei diesem Sprachkonzept keinen Sinn.

Grundlegendes Prinzip von Forth ist das Operieren mit dem Stapelspeicher (Stack). Dieser funktioniert nach dem LIFO-Prinzip (Last In, First Out). Alle Werte, die auf dem Stapel abgelegt wurden, lassen sich nur in umgekehrter Reihenfolge wieder herunternehmen. Für einen problemlosen Ablauf dieses Systems sorgen eine ganze Reihe von Stack-Befehlen, mit denen sich Werte verschieben und vertauschen lassen. Sämtliche mathematischen Operationen laufen in Forth über den Stack. Man bedient sich hierbei der Umgekehrten Polnischen Notation (UPN), die recht gewöhnungsbedürftig und Benutzern von HP-Taschenrechnern bekannt ist.

Forth ist ein sehr offenes System und durch seine Assemblernähe universell einsetzbar. Fehlende Funktionen können jederzeit selbst programmiert werden. Zudem ist Forth sehr schnell. Das Erlernen der Sprache und die Übersichtlichkeit der Programme können als noch ausreichend eingestuft werden. Auf jeden Fall fasziniert Forth jeden, der sich länger damit beschäftigt. Für alle verbreiteten Mikrocomputer existieren mittlerweile eine oder mehrere Versionen.

Logo — kinderleicht

Seymour Papert, der als geistiger Vater der Sprache Logo gilt, arbeitete 12 Jahre an der Verwirklichung dieser »Erziehungsphilosophie«. Er leitete ein eigens gegründetes Entwicklungsteam aus Programmierern und Lehrkräften am MIT (Massachusetts Institute of Technology) in den USA. Man arbeitete damals ausschließlich auf den größten vorhandenen Datenverarbeitungsanlagen. Dadurch fand ein wesentliches Konzept der KI-Sprache Lisp in Logo Anwendung: die Listenprogrammierung. Listen sind einfach zu definieren und können per Befehl manipuliert, kombiniert und verglichen werden. Eine leicht programmierbare Dateiverwaltung ist nur ein Anwendungsbeispiel dieser Technik.

Bekannt wurde Logo vor allem durch die Schildkröte, ein kleines Zeichensymbol der »Turtle-Graphics«. Mit ihr lassen sich auf einfache Weise die tollsten Grafiken zaubern. Die Schildkröte krabbelt über den Bildschirm und hinterläßt dabei eine sichtbare Spur.

Eine Schildkröte machte Logo bekannt

Gesteuert wird mit einfachen Befehlen wie FORWARD, BACK, LEFTTURN, RIGHTTURN. Zusätzlich muß noch die Länge der zurückgelegten Strecke und des Drehwinkels angegeben werden. Ebenso ist eine Standortabfrage der Turtle möglich. Logo-Programme ähneln in der Struktur dem Baukastenprinzip der Forth-Programme. Mit Hilfe des Interpreters lassen sich einzelne Bausteine erproben und später zum eigentlichen Programm zusammensetzen. Der Komfort ist dabei in Logo ungleich höher als in allen bisher genannten Sprachen. So kann der Anwender vorerst Begriffe wie Archivierung, Dateien und andere spezielle Funktionen der Datenverarbeitung links liegen lassen. Diese Vorteile gehen aber leider zu Lasten des Speicherplatzes.

Eng mit dem Prozedurkonzept verbunden ist die Rekursivität von Logo. Prozeduren sind in der Lage, sich selbst aufzurufen. Auf diese Weise lassen sich schnell reizvolle grafische Gebilde erzeugen und gewisse mathematische Zusammenhänge einfach ausdrücken. Rekursive Strukturen sind in Basic gar nicht und in vielen anderen Sprachen nur mit hohem Aufwand zu verwirklichen.

In die Logo-Philosophie wurden Erziehungstheorien des Schweizer Philosophen Jean Piaget eingebracht. Dieser hatte zuvor das Lernverhalten von Kindern analysiert. Tatsächlich wirkt Logo besser auf die Denkweise eines Schülers als Basic oder Pascal. Bemängelt werden muß bei Logo hauptsächlich die geringe Verarbeitungsgeschwindigkeit der Programme. Sie fällt aber bei einem Lernsystem nicht so stark ins Gewicht.

Wegen des hohen Speicherbedarfs sind Logo-Interpreter auf Mikrocomputern nur als Ausschnitt des Gesamtsystems erhältlich. Dies wird sich mit wachsendem Speicherstandard jedoch bald ändern.

Comal — gelungene Essenz

Im Jahre 1973 ging Comal aus den Sprachen Basic und Pascal als ein neuer Ableger hervor. Später kamen in Comal noch Elemente von Logo hinzu, so zum Beispiel die Schildkrötengrafik. Zudem sind in Comal der Compiler und der Interpreter nicht getrennt vorhanden, sondern es wurden deren beste Elemente in einer Zwischenstufe zusammengefaßt. Ein Comal-Programm besteht aus drei Schritten. Im ersten wird schon bei der Programmeingabe die Syntax überprüft. Dieser Syntaxchecker ist selbst für den ohnehin schon eingabefreundlichen Interpreter ungewöhnlich komfortabel. Die Comal-Schlüsselworte werden sofort in sogenannte »Token« übersetzt, das sind Abkürzungen, die nur ein Byte beanspruchen. Dieses Prinzip verwenden übrigens alle Interpreter. Der zweite Schritt beginnt nach dem Programmstart. In einer Art Compilerdurchlauf wird der Programmtext nach Variablen, Prozeduren, Funktionen und Sprüngen durchsucht. Die Ergebnisse dieser Analyse werden dann in einer gesonderten Liste zusammengefaßt. Man kann diesen Vorgang auch als eine Art automatische Deklaration ansehen. Im dritten Schritt, dem Programmlauf selbst, wird auf diese Liste ständig direkt zugegriffen. So ergeben sich gegenüber Basic, wo der Interpreter oft den ganzen Programmtext absucht, enorme Geschwindigkeitsvorteile.

Die Comal-Syntax lehnt sich stark an die von Basic an. Im Direktmodus wurden die meisten Befehle Wort für Wort übernommen. Das gleiche gilt für einige Befehle des Programm-Modus. Einige Comal-Versionen akzeptieren neben den eigenen Schlüsselworten sogar noch gleichbedeutende Basic-Befehle. Wer aus Basic zu Comal aufsteigt, ist so zwar vor Irrtümern einigermaßen sicher, jedoch wird dem Prinzip der Eindeutigkeit nicht gerade Rechnung getragen.

Von Pascal wurde die Strukturiertheit übernommen, dessen strenge Syntaxvorschriften aber erfreulicherweise vermieden. Die typischen Kontrollstrukturen, die schon bei Pascal und Basic beschrieben wurden, sind ausnahmslos vorhanden. Die Lesbarkeit des Pro-

grammtextes wird zudem dadurch unterstützt, daß Comal beim Listen eine optische Gliederung vornimmt.

Comal wird für fast alle gängigen Mikrocomputer angeboten. Erfreulicherweise stehen auch, ähnlich wie bei Forth, zahlreiche Public-Domain-Versionen zur Verfügung. Diese unterscheiden sich von kommerziellen Sprachangeboten lediglich im Umfang. Es wird erwartet, daß Comal in Zukunft eine starke Verbreitung erfährt. Jüngster Anhaltspunkt für diese These ist der Beschluß der Kultusministerien, Pascal im Informatikunterricht allmählich durch Comal zu ersetzen.

C – die Zukunft?

»C« – für diesen einen Buchstaben lassen viele Programmierer Pascal, Forth oder Fortran links liegen. Viele betrachten C als die Programmiersprache schlechthin. Tatsächlich bietet C viele bestechende Vorteile, die sich von denen der anderen Hochsprachen unterscheiden. In C wurden bekannte Betriebssysteme wie Unix und GEM geschrieben.

Die Entwicklung von C reicht bis an den Anfang der siebziger Jahre zurück. Die Namensgebung war ebenso kurios wie einfallslos. 1970 begann die Firma Digital Equipments mit der Entwicklung von Spezialsprachen, die die Programmierung von Minicomputern unterstützen sollten. Diese wurden einfach nach dem Alphabet benannt. B war also eine Weiterentwicklung von A und diente 1971 dazu, Unix auf verschiedene Rechner zu übertragen. Bald darauf erkannten Dennis Ritchie und Ken Thompson, die damals bei Bell Laboratories beschäftigt waren, die Leistungsfähigkeit von B. Sie verbesserten diese Sprache bis 1973 entscheidend. Wie das Endprodukt heißt, wissen wir bereits. Unix wurde wenig später ebenfalls auf C umgeschrieben.

Wenn ganze Betriebssysteme in C gehalten sind, läßt sich die ungeheuere Effizienz dieser Sprache schon erahnen. Das C-Compilat geht mit dem Speicher äußerst sparsam um, und ist zudem um den Faktor 50 schneller als vergleichbare Basic-Programme.

Der eigentliche Befehlsvorrat von C ist denkbar klein. Er umfaßt nur 13 Instruktionen. Bei der Erzeugung eines C-Programms wird der Quelltext zuerst mit dem mitgelieferten Editor oder einer Textverarbeitung geschrieben. Dieser dient dann als Eingabedatei für den Compiler. Bis hierher besteht also kein Unterschied zur Vorgehensweise mit den meisten anderen Compilersprachen. Der Compiler überprüft den Text und übersetzt diesen mit den ihm bekannten Befehlen in eine Zwischendatei und legt diese auf einen externen Speicher (zum Beispiel Diskette) ab. Anschließend beginnt das sogenannte »Linking«. Der Linker ordnet aus einer externen Bibliothek die noch fehlenden Befehle entsprechendem Assemblercode zu. So entsteht schließlich das lauffähige Programm, das fast so kompakt ist, als wäre es direkt in Maschinensprache geschrieben. Das Compilieren benötigt jedoch wegen des Zwischencodes je nach Geschwindigkeit des Speichermediums mehr Zeit, als die unmittelbare Übersetzung in Maschinensprache.

Die Vorteile des Compiler-Linker-Prinzips wiegen diesen Nachteil bei weitem auf: C ist praktisch uneingeschränkt portabel und die Linker-Funktion kann man selbst erweitern. C-Programme bieten neben der selbstverständlichen Strukturierung einige ungewöhnliche aber vorteilhafte Eigenschaften. So sind in Anlehnung an Assembler Befehle zum Inkrementieren und Dekrementieren vorhanden, Variable können als Registervariable deklariert werden. Daß die Benutzung derartiger Sprachmittel einem Compiler das Leben leicht macht, ist klar. Einschränkungen entstehen jedoch bei Prozessoren, die nur wenige Register aufweisen, wie beispielsweise die drei 8-Bit-Register des 6502, der nur A, X und Y zu bieten hat.

Ein weiteres nennenswertes Stilmittel unter C sind Makros. Diese sind mit Unterprogrammen vergleichbar, die nicht angesprungen werden müssen, sondern jeweils erneut vom Compiler in den Objektcode eingebunden werden. Diese Technik bringt einen großen Zeitvorteil, da Parameterübergaben und Sprünge entfallen.

Im 8-Bit-Bereich ist C nur auf dem Z80 unter CP/M verbreitet. Eine Version für den C 64 stellt hier eine erfreuliche Ausnahme dar. Für 16- und 32-Bitter existieren aber umfangreichere Versionen.

Ada – gekrönter Adel

Ada ist heute als Krönung bei der Entwicklung modularer Programmiersprachen anzusehen. Die Sprache wurde erst in jüngster Zeit im Auftrag des weltweit größten Softwaresponsors entwickelt, dem Pentagon. Der finanzielle und organisatorische Aufwand dafür war entsprechend riesig. Benannt ist die Sprache nach der jungen Gräfin Ada Byron, die um 1830 für Babbages (siehe oben) Rechenmaschinen ein nahezu komplettes Programm zur Berechnung der Bernoullischen Zahlen schrieb.

Der Aufwand, der um Ada getrieben wurde, erklärt sich mit einer Kalkulation des US-Verteidigungsministeriums. Danach können zwischen 1983 und 1999 etwa 24 Milliarden Dollar (!) eingespart werden, wenn eine einzige universelle Programmiersprache die bisherigen 450 (!!) Programmiersprachen ersetzen könnte. Ada ist ähnlich PL/1 sehr umfangreich. Es wäre daher zwecklos, auf einzelne Sprachelemente einzugehen. Deshalb hier nur die grundsätzlichen Sprachkonzepte von Ada:

- Das Modulkonzept von Ada ist äußerst umfangreich. Es stehen sowohl datenorientierte als auch funktionsorientierte Module zur Verfügung. Innerhalb der datenorientierten Pakete lassen sich fast beliebige Datentypen und Datenstrukturen realisieren.
- Ähnlich wie in Pascal können Datenstrukturen geschachtelt werden. Umfangreichere Prozedur- und Funktionskörper werden ausgelagert, zum Beispiel um die Lesbarkeit der Programme zu erhöhen.
- Sämtliche Kontrollstrukturen, von UNTIL bis hin zu CYCLE-Schleifen stehen zur Verfügung. Ferner sind alle linearen und strukturierten Datentypen implementiert.
- Ein automatischer Textformatierer (Pretty-Printer) wertet Schachtelungsstrukturen aus und sorgt für ein übersichtliches Layout des Programmtextes.
- Neben Parallelverarbeitung (Multitasking) gehören Konzepte wie die Parallel- und Ausnahmebehandlung zu den bemerkenswerten Fähigkeiten, auf deren Erklärung hier wegen der komplexen Zusammenhänge verzichtet wird.

Im großen und ganzen wurden die gesetzten Ziele bei der Entwicklung von Ada nach dem heutigen Erkenntnisstand optimal erreicht. Die zu Anfang unter den Qualitätsaspekten genannten Stichworte wie Vollständigkeit, Zuverlässigkeit, Korrektheit, Übertragbarkeit, Wartung und Fehlerbehandlung wurden weitestgehend realisiert. Die Einfachheit der Sprache ist als ausreichend zu betrachten, wenn auch die vollständige Einarbeitung in dieses System Jahre beansprucht. Ada ist für Mikrocomputer zur Zeit nur unter MS-DOS verfügbar und auch hier nur in einer abgespeckten Version.

Wir sind nun mit dem Aufstieg im »modernen Turm zu Babel« fast an der Spitze angelangt. Nach unten blickend können wir die gebräuchlichsten Sprachen beurteilen.

(Matthias Rosin/ev)

Die C-Crew im Test



C-Compiler sind in aller Munde, und die Beliebtheit der Sprache C steigt ständig. Steigen auch Sie ein in diese Supersprache. Wir zeigen Ihnen den richtigen C-Compiler für den Atari ST.

Das Angebot an C-Compilern für den Atari ST wächst: Kein Wunder, denn C ist eine Sprache, die das Betriebssystem des Atari ST voll unterstützt. Das Betriebssystem wurde auch in dieser Sprache verfaßt.

Da Softwarehäuser C-Programme wegen ihrer Portabilität sehr schätzen, ist im Lieferumfang des Entwicklungspakets zum Atari ST auch der C-Compiler von Digital Research enthalten.

Dieser Compiler, als Teil eines Profi-Systems, hat zwar noch einige Fehler, aber es läßt sich vernünftig damit arbeiten. Den Heimanwender interessiert das System wegen des hohen Preises wohl weniger. Atari gibt den Compiler leider nur mit dem Entwicklungspaket ab - einzeln ist er nicht zu beziehen.

Es gibt aber inzwischen gute Alternativen. Allerdings ist wegen des begrenzten Angebots die Auswahl noch nicht sehr groß; doch einige Firmen kündigten bereits eigene Compiler an, und allzulange werden diese wohl auch nicht mehr auf sich warten lassen. Für kommende rosige Zeiten unterbreiten wir Ihnen hier einige Auswahlkriterien für C-Compiler.

Als erstes sollten Sie darauf achten, wie vollständig die Sprache implementiert ist. Mit allen 28 Schlüsselwörtern ist der Sprachumfang vollständig. Viele Entwickler verzichten jedoch auf das eine oder andere Leistungsmerkmal, und da muß man sich genau überlegen, inwieweit das für die eigenen Zwecke tragbar ist.

Erfahrungsgemäß wächst mit dem Wissen auch der Anspruch. Schnell stellt man fest, daß der Compiler, der noch vor kurzem genügte, für die nächste Anwendung große Mängel aufweist. Man sollte sich vorher genau über das Produkt informieren und eventuell etwas tiefer in die Tasche greifen, um vor einer Enttäuschung sicher zu sein.

Eine »Einsteigerversion« ist der C-Compiler von GST. Er kann nur ganze Zahlen verarbeiten und führt keine Fließkomma-Arithmetik durch. Damit beschränkt sich sein Einsatzgebiet auf Systemprogrammierung und weitere Anwendungsprogramme, wie Textverarbeitung, Datenbanken und Spiele.

Natürlich kann man sich die Floatingpoint-Routinen auch selbst schreiben, aber es gibt bereits andere C-Compiler für den Atari-ST, die nicht viel mehr kosten und bereits alles beinhalten. Beim GST-Compiler verzichtete man auch auf die Strukturen (das Schlüsselwort »structure« fehlt). Um gute Algorithmen zu schreiben, ist man aber auf Datenstrukturen, und damit auf dieses Schlüsselwort angewiesen.

Kompletter Sprachumfang

Wer etwas tiefer einsteigen möchte in die faszinierende Programmiersprache C, sollte lieber zum Compiler von Lattice greifen. Er ist der einzige, der den kompletten Sprachumfang von Kernighan und Ritchie besitzt. Beim DRI-Compiler arbeiten die Funktionen scanf, für formatierte Ausgabe, und getchar, um ein Zeichen einzulesen, nicht korrekt. Der Lattice-Compiler hat aber wiederum andere kleine Fehler. Beim Arbeiten mit Fenstern macht er Schwierigkeiten, da er eine falsche »Window_handle«-Nummer zurückgibt. Ansonsten darf man ihn getrost als die zur Zeit beste Implementation für den ST bezeichnen.

Eines der wichtigsten Kriterien für einen C-Compiler sind seine Libaryfunktionen.

Zum Standard-C gehört eine Standard-Libary. Informationen über den Umfang erhält man aus der Sprachbeschreibung von Kernighan und Ritchie (siehe Literaturverzeichnis), die auch den C-Standard definierte.

Dieser sogenannte K&R-Standard sollte in der Libary auf keinen Fall fehlen.

Wichtig ist außerdem, wie die Zugriffe auf GEM und das Betriebssystem geregelt sind. Für die GEM-Programmierung selbst bietet der GST-Compiler am meisten an. Bei den Betriebssystemaufrufen macht er leider große Abstriche.

Die Libary enthält keine der Funktionen »gemdos«, »bios« und »xbios«, sondern lediglich eine kleine Auswahl vordefinierter »gemdos«-Funktionen. Für den Profi ist das kein Problem: Mit nur wenigen Assembler-Kenntnissen kann man sich die fehlende Betriebssystem-Schnittstelle selber stricken. Dennoch, auch diese Einschränkung sollten Sie bei einer Kaufentscheidung berücksichtigen.

Hier schneidet wiederum das Lattice-Produkt am besten ab. Seine Libary-Funktionen sind außerordentlich umfangreich.

Zwar nicht ausschlaggebend, aber erwähnenswert, bleiben die Extras, wie zum Beispiel ein Editor. Der GST-Compiler zeichnet sich durch einen sehr guten Editor aus. Entscheidender aber sind Hilfen zur Fehlersuche.

Nun kennen Sie alle zur Beurteilung eines C-Compilers wichtigen Kriterien. Haben Sie alle für Sie wichtigen Punkte beachtet, dann lassen Sie sich von der Leistungsfähigkeit Ihres C-Compilers überraschen. (hb)

Info: C-Compiler von Digital Research Inc. Atari Corp. (Deutschland) GmbH, Frankfurter Str. 89-91, 6096 Raunheim, Tel: 06142/41081. Nur erhältlich mit dem Entwicklungspaket für den Atari ST, Preis 969 Mark.

C-Compiler von GST in Kürze bei Atari erhältlich. Preis zirka 300 Mark.

Lattice-C-Compiler von Metacomco. Vertrieb über Philgerma, Ungererstr. 42, 8000 München 40, Tel. 089/395551, Preis 380 Mark.

Small-C – ein C-Compiler unter CP/M

C wird in Zukunft immer mehr an Bedeutung gewinnen. C verbindet die Eigenschaften einer Hochsprache mit denen einer Maschinensprache. C-Programme sind sehr schnell. Drei gute Gründe für C! Unter CP/M gibt es jetzt einen preiswerten Einstieg.

C ist eine Sprache, die in letzter Zeit an Bedeutung gewinnt. Das berühmte Betriebssystem Unix ist in C geschrieben. Und immer mehr kommerzielle Software wird in C entwickelt. Es gibt einige C-Compiler, die unter dem Betriebssystem CP/M laufen. Stellvertretend für diese präsentieren wir Small-C, ein Entwicklungssystem, das eine umfangreiche Programmbibliothek (Quellcode - ebenfalls in C) mitbringt.

Das Programmpaket Small-C enthält außer dem C-Compiler noch einen Makro-Assembler zur Erzeugung von relokatierbarem 8-Bit-Objektcode für 8080/Z80-Prozessoren und einen Linker zum Binden einzelner Module zu lauffähigen Programmen. Außerdem bereichern noch eine Menge hilfreiche und gleich mitgelieferte Dienstprogramme (Tools) dieses Paket. Als Hardware-Voraussetzung benötigt man das Betriebssystem CP/M und mindestens 56 KByte Hauptspeicher.

Der Commodore 128 im CP/M-Modus findet in diesem Entwicklungssystem ein geeignetes Werkzeug, um mit der Sprache C die ersten Gehversuche zu unternehmen. Aber auch für alte Hasen liefert es eine Menge Anregungen zum Programmieren. Das gesamte Paket ist bis auf ein paar arithmetische und logische Funktionen selbst in C geschrieben.

C in C geschrieben

Der C-Compiler erzeugt aus einem C-Quellcode einen Assembler-Quellcode und erst dieser wird in einen relokatierbaren Objektcode übersetzt. Dazu eignet sich der mitgelieferte Small-MAC-Assembler oder der M80-Assembler von Microsoft. Der Sprachumfang umfaßt lediglich eine Untermenge der Sprachdefinition nach Brian W. Kernighan und Dennis M. Ritchie, den Entwicklern der C-Sprache. Die Definitionen, Fließkomma-Datentypen (float,double), sizeof, mehrdimensionale Arrays, Zeigerarray, Strukturen (struct,union), Bit-Felder und casts fehlen. Daß aber trotzdem größere Programme erzeugt werden können, zeigt das Small-C-Paket selbst. Es sind alle Unix-Funktionen implemetiert, soweit sie in einer fremden Umgebung anwendbar sind.

Der Small-MAC-Assembler beinhaltet ein ganzes Paket von Programmen. Er enthält im einzelnen einen Makro-Assembler (MAC), einen Linker (LNK) und einen Bibliotheksverwalter (LIB). Zusätzlich stehen noch ein Lader (LGO), ein CPU-Anpassungsprogramm (CMIT) und ein Dump-Programm (DREL) zur Verfügung. Der Makro-

Ein-/Ausgabe

fopen(name, mode)
freopen(name, mode, fd)
fclose(fd)
fgetc(fd)
ungetc(c, fd)
getchar()
fgets(str, sz, fd)
fread(ptr, sz, cnt, fd)
read(fd, ptr, cnt)
gets(str)
feof(fd)
ferror(fd)
clearerr(fd)
fputc(c, fd)
putchar(c)
fputs(str, fd)
puts(str)
fwrite(ptr ,sz, cnt, fd)
write(fd, ptr, cnt)
fflush(fd)
cseek(fd, offset, from)
rewind(fd)
ctell(fd)
unlink(name)
rename(old, new)
auxbuf(fd, size)
iscons(fd)
isatty(fd)
printf(str, arg1, ...)
fprintf(fd, str, arg1, ...)
scanf(str, arg1, ...)
fscanf(fd, str, arg1, ...)

Formatkonvertierung

atoi(str)
atoib(str, base)
itoa(nbr, str)
itoab(nbr, str, base)
dtoi(str, nbr)
otoi(str, nbr)
utoi(str, nbr)
xtoi(str, nbr)
itod(nbr, str, sz)
itoo(nbr, str, sz)
itou(nbr, str, sz)
itox(nbr, str, sz)

Zeichenketten

left(str)
pad(str, ch, n)
reverse(str)
strcat(dest, sour)
strncat(dest, sour, n)
strcmp(str1, str2)
lexcmp(str1, str2)
strncmp(str1, str2, n)
strcpy(dest, sour)
strncpy(dest, sour, n)
strlen(str)
strchr(str, c)
strrchr(str, c)

Zeichenklassifizierung

isalnum(c)
isalfa(c)
isascii(c)
iscntrl(c)
isdigit(c)
isgraph(c)
islower(c)
isprint(c)
ispunct(c)
isspace(c)
isupper(c)
isxdigit(c)
lexorder(c1, c2)

Zeichenumwandlung

toascii(c)
tolower(c)
toupper(c)

mathematisch

abs(nbr)
sign(nbr)

Programmkontrolle

calloc(nbr, sz)
malloc(nbr)
avail(abort)
free(addr)
getarg(nbr,str,sz,argc,argv)
poll(pause)
exit(errkode)

Tabelle 1. Vollständiges Verzeichnis der Funktionen von Small-C

CHG(Change)	Ersetzen von Zeichenketten in Textdateien
CNT(Count)	Zählen von Zeichen, Wörtern oder Zeilen
CPY(Copy)	Kopieren von Textdateien
CPT(Crypt)	Ver- und entschlüsseln von Dateien
DBT(Detab)	Ersetzen von Tab-Zeichen durch Leerzeichen
EDT(Edit)	Zeileneditor
ETB	Gegensatz von DBT
FND(Find)	Suchen von Zeichenketten in Textdateien
FNT	Auswahl von Schriftarten des Epson-FX-80 und kompatible Drucker
FMT(Format)	Formatieren (Druck aufbereiten) von Textdateien
LST(List)	Ausgabe von Textdateien auf den Bildschirm
MRG(Merge)	Zusammenhängen von zwei sortierten Textdateien
PRT(Print)	Drucken von Textdateien
SRT(Sort)	Sortieren von Textdateien
TRN(Trans)	Kopiert Textdateien und ändert Zeichenketten

Tabelle 2. Dienstprogramme für Textverarbeitung und Dateiverwaltung

Assembler MAC ist ein 2-Pass-Compiler zur Erzeugung von verschiebbarem Objektcode. Der erzeugte Code liegt im Microsoft-8-Bit-Format zur Weiterverarbeitung mit dem Linker LNK oder L80 von Microsoft vor. Diese Linker verknüpfen einzeln übersetzte Module oder solche, die in einer Bibliothek stehen, zu einem lauffähigen Programm, einer sogenannten COM-Datei. Auch der Lader ist ein nützliches Hilfsmittel. Er ermöglicht es, Programme an eine bestimmte Adresse in den Speicher zu laden und wahlweise zu starten. Damit werden Betriebssystemerweiterungen beim Booten (Kaltstart) installiert. Der Bibliotheksverwalter (LIB) verwaltet eine Sammlung von verschiebbaren Objektmodulen. In der Tabelle 1 sehen Sie eine Übersicht aller vorhandenen Funktionen. Die Verwaltung geschieht mit Hilfe einer Indexdatei. In dieser Datei stehen die Namen und die Adressen der Module aus der Bibliothek. Wird nun ein Programm mit dem Linker zusammengebunden, so sucht der Computer zunächst den Namen in der Indexdatei. Bei einem positiven Suchergebnis wird das Objektmodul aus der Bibliothek an das lauffähige Programm angehängt. Der Assembler arbeitet mit Maschinen-Instruktions-Tabellen, um einen Quellcode zu übersetzen. Um den Assembler für verschiedene Prozessoren (8080/Z80) zu verwenden, muß ihm eine solche Tabelle zur Verfügung stehen. Das Programm CMIT paßt den Small-MAC-Assembler einem bestimmten Prozessor an, indem er eine Tabelle (es sind für beide Prozessortypen die Tabellen in Quellform vorhanden) in ein internes Format verwandelt und dieses in den ausführbaren Small-MAC-Assembler kopiert. Das Hilfsmittel DREL erzeugt aus jedem Objektmodul eine Liste in hexadezimalem Format. Die Ausgabe erfolgt im Standardformat. Es kann damit nach Belieben in eine Datei, auf einen Drucker oder auf den Bildschirm ausgegeben werden.

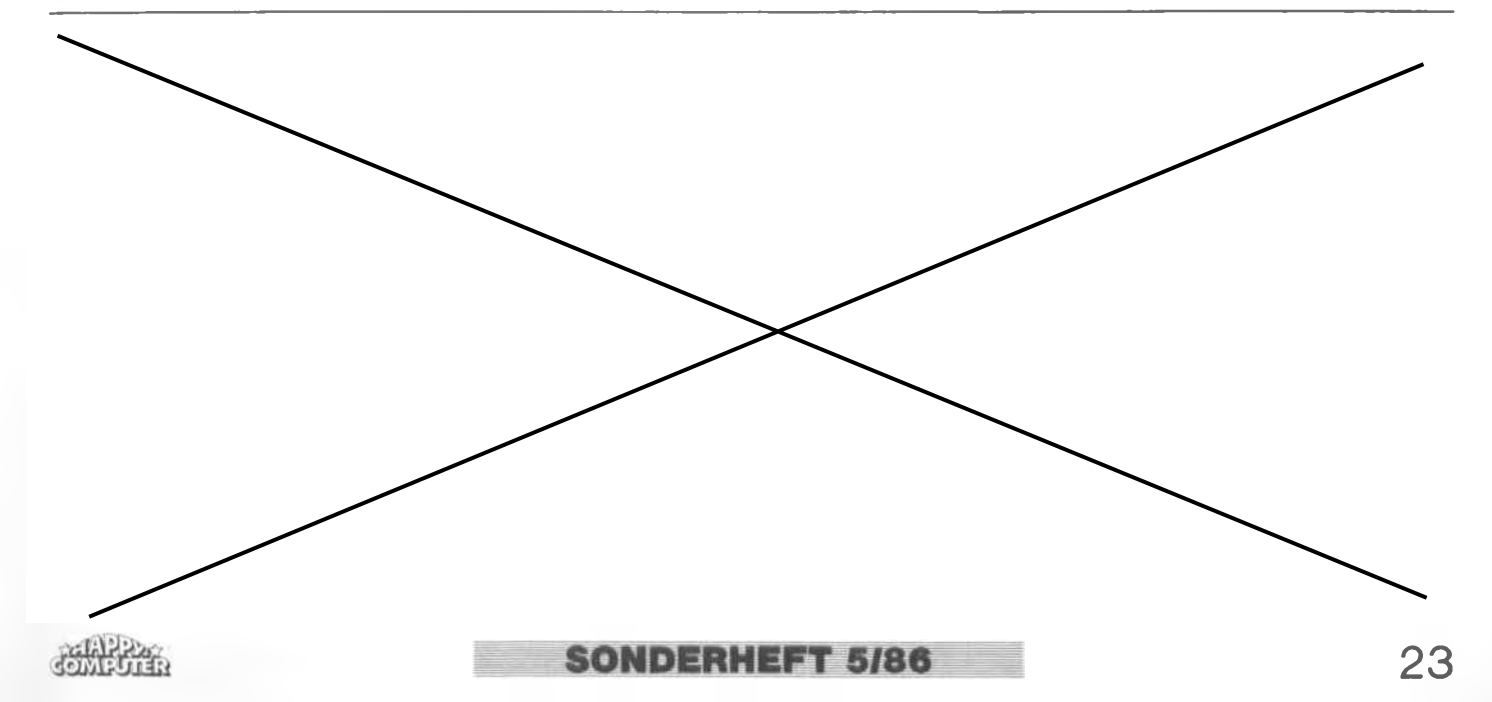
Neben dem C-Compiler und dem Assembler enthält das Entwicklungssystem noch viele andere nützliche Programme. Sie umfassen folgende Funktionen und können nach Belieben vom Anwender erweitert werden:
- editieren, formatieren, sortieren, zusammenfügen, listen, drucken, suchen, ersetzen, übersetzen, kopieren und aneinanderfügen, verschlüsseln und entschlüsseln, Leerzeichen durch Tabs ersetzen, Tabs durch Leerzeichen ersetzen, Zeichen, Wörter und Zeilen zählen, Druckerzeichensatz auswählen.

Tabelle 2 listet alle vorhandenen Programme auf. Die Dienstprogramme (insgesamt 15) werden nur als Quelldatei geliefert. Sie müssen also erst mit Small-C übersetzt und dann assembliert und zusammengefügt werden.

Außer den Quellen der Programme befinden sich noch verschiedene Include-Dateien auf den insgesamt drei Disketten, die man beim Kauf bekommt. Alles in allem beläuft sich die Anzahl der Programme auf den Disketten auf zirka 100 Dateien. Das deutsche Handbuch ist mit 200 Seiten sehr umfangreich und geht auf alle Programme sehr ausführlich ein. Die umfassenden Kommentare bewahren auch den Anfänger vor großen Problemen.

Das Entwicklungssystem Small-C bietet, trotz des eingeschränkten Sprachumfangs und seiner durch das Betriebssystem CP/M bedingten langsamen Diskettenverarbeitung, ein sehr gutes Preis/Leistungsverhältnis. Es gewährt vor allem dem C-Einsteiger hilfreiche Unterstützung beim Programmieren beziehungsweise beim Erlernen von C. Das Small-C wird zusammen mit C-Quellcode, Editor, Assembler, Linker und Tools zur Textverarbeitung für den C 128 und 128 D, den Schneider CPC 464/664/6128 und den Joyce zum Preis von 148 Mark von Markt & Technik angeboten. Besitzer des CPC 464/664 benötigen allerdings noch eine Speichererweiterung.

(Günter Langheinrich/cg)



C – wie Cäsar

Der römische Imperator war der mächtigste Mann seiner Zeit. C herrscht genauso in der Welt der Programmiersprachen. Lernen Sie die faszinierende Sprache kennen.

C ist eine noch relativ junge Programmiersprache, die Anfang der siebziger Jahre von Dennis M. Ritchie entwickelt wurde. Er arbeitete damals in den Bell Laboratories in den USA und stand vor dem Problem, ein komplettes Betriebssystem (und zwar Unix) in Assembler zu schreiben. Bis zu diesem Zeitpunkt kannte man keinen anderen Weg Betriebssysteme zu entwickeln. Die Mikroprozessoren waren noch sehr langsam und Speicherbausteine teuer.

Doch Dennis Ritchie waren die Nachteile von Maschinencode klar: Solche Programme sind nur schwer auf andere Mikroprozessoren zu übertragen. Selbst die Adaption auf andere Computer mit dem gleichen Prozessor führt zu kaum überwindbaren Schwierigkeiten. Außerdem sollte Unix alle vorherigen Betriebssysteme in Bezug auf Leistungsfähigkeit in den Schatten stellen. Das bedeutete aber auch, daß der Umfang des Programmcodes gigantisch werden mußte.

Ab einer Länge von mehreren KByte sind Assemblerprogramme praktisch nicht mehr überschaubar und enthalten mit Sicherheit versteckte Bugs (bösartige Programmierfehler), die oft erst nach jahrelangem Einsatz des Systems zum Vorschein kommen. Noch heute gibt es in CP/M 2.2 kleine Fehler, die nie beseitigt wurden. Und das, obwohl die erste Version von CP/M immerhin im Jahr 1973 erschien.

So sah sich Ritchie nach einer Programmiersprache um, die sich zur Systemprogrammierung eignet. Da sich aber alle »konventionellen« Sprachen der damaligen Zeit – zum Beispiel Fortran, Algol und Cobol – als unbrauchbar erwiesen, entschloß er sich, eine neue Sprache für seine Zwecke zu schreiben. Am Ende dieser Entwicklung stand C in der Form, in der wir es heute kennen. Und Ritchie hat tatsächlich das Meisterstück vollbracht, etwa 95 Prozent von Unix in C zu programmieren. Eine beeindruckende Leistung, denn »seine« Programmiersprache ist schnell genug, mehrere Benutzer zu bedienen und mehrere Prozesse (Programme) gleichzeitig zu bearbeiten.

Natürlich stammt nicht alles zu 100 Prozent vom Entwickler selbst. Er griff auch auf ältere Sprachen zurück. Die »Vorfahren« von C sind CPL, BCPL und B – Programmiersprachen, die heute fast niemand mehr kennt. CPL, die »Combined Programming Language«, wurde zu Beginn der sechziger Jahre als eine Gemeinschaftsarbeit der Universitäten von Cambridge und London entwickelt. Aber: Viele Köche verderben den Brei. Das Resultat war eine Monstersprache, die den meisten Programmierern zu schwerfällig und zu undurchsichtig war. Rettung nahte, als Martin Richards in Cambridge begann, sich für die Sprache zu interessieren. Er kürzte alles Entbehrliche heraus, und fertig war die neue Sprache BCPL, die »Basic Combined Programming Language«. Eines der heute bekanntesten Produkte in BCPL dürfte der Basic-Interpreter für die ST-Computer sein.

Zur Systemprogrammierung besser geeignet, weil noch knapper gehalten, ist B. Die Sprache mit dem kurzen Namen stammt von Ken Thompson, der sie im Jahr 1970 im Auftrag der Bell Laboratories entwickelte. B besitzt nur einen einzigen Datentypen, das Maschinenwort. Das war Dennis Ritchie dann aber doch zu wenig, und als er sich für Unix zwischen B und Assembler entscheiden sollte, wählte er die – damals erst theoretisch existierende – Sprache C. C baut auf B auf, ist aber entschieden Programmierer-freundlicher. Nicht zuletzt diese Eigenschaft dürfte erheblich zu ihrer Verbreitung beigetragen haben.

Warum C?

Heute ist C auf dem besten Weg, sowohl Hochsprachen – wie Pascal und Fortran – als auch die diversen Maschinensprachen zu verdrängen. Die Entwicklung geht inzwischen dahin, nur noch einen C-Compiler für neue Prozessoren und Betriebssysteme in Assembler zu schreiben und die weitere Software mit dem C-Compiler zu erzeugen. Bei 16-Bit-Prozessoren mit 8 MHz Taktfrequenz –wie dem Motorola 68000 im Atari ST oder dem Amiga von Commodore – kann man sich den unvermeidlichen Geschwindigkeitsverlust durchaus leisten.

C bietet Kontrollstrukturen, die mit denen in Pascal, dem »Musterschüler« der strukturierten Programmierung, durchaus auf eine Stufe zu stellen sind. Pascals CASE heißt in C SWITCH CASE, REPEAT UNTIL und WHILE nennen sich DO WHILE und WHILE. Auch die allseits beliebte FOR-Schleife ist – wenn auch mit geänderter Syntax – wiederzufinden. Andererseits hat der Programmierer in C alle Möglichkeiten, auch auf der untersten Hardware-Ebene, direkt mit den Peripheriegeräten, zu arbeiten.

Eine der schönsten Eigenschaften von C haben wir aber beinahe vergessen: »Portabilität« heißt das Zauberwort. Wer von einem Computer auf einen anderen umsteigt, kann praktisch die gesamte C-Software mitnehmen. Diese Kompatibilität der einzelnen C-Versionen erreicht keine andere Sprache, nicht einmal das vielgerühmte Pascal. Versuchen Sie einmal, ein Programm, das starken Gebrauch von den Fähigkeiten des Turbo-Pascals macht, auf MS-Pascal oder Pascal/MT+ zu übertragen. Von den Hunderten von Basic-Dialekten wollen wir gar nicht erst reden.

Was für Hobby-Programmierer gilt, trifft für Software-Firmen, die Programme kommerziell vertreiben, noch viel mehr zu. Ein großer Teil der Produktionskosten für Software läßt sich einsparen, wenn Programme in kürzester Zeit auf anderen Computern zum Laufen gebracht werden können. Die Software-Entwicklung wird damit von den Fortschritten der Hardware unabhängiger.

Derart motiviert, können Sie sich sicher nicht mehr dem Bann entziehen, der von C ausgeht. Es stellt sich nur noch ein »kleines« Problem, bevor Sie mit dem Programmieren beginnen können: Sie brauchen einen Compiler. Für MS-DOS-Computer gibt es inzwischen fast mehr C- als Pascal-Compiler. Im Bereich der Heim-Computer ist das Angebot deutlich geringer. Für den Sinclair Spectrum und den Schneider-CPC bietet HiSoft C-Compiler an (wobei die Schneider-Version erheblich leistungsstärker ist). Für den Commodore 64 gibt es einen Compiler von Data Becker. Allerdings ist der 6502 für die Implementation eines C-Compilers und für die Bearbeitung von C-Programmen denkbar ungeeignet. Die 6502-CPU ist technisch veraltet und bietet mit drei Registern (Akku, Indexregister X und Y) nicht genug für eine so Register-intensive Sprache wie C. Wer einen Commodore 128 hat, sollte sich deshalb entschließen, auf CP/M umzusteigen.

Unter CP/M nämlich ist das Angebot reichhaltig. Ein großer Teil der Compiler kann aber auf dem CPC 464 und CPC 664 von Schneider ohne Speichererweiterung nicht zum Laufen gebracht

werden. Am besten stehen somit wieder einmal die Besitzer des CPC 6128 da, die über CP/M 3.0 mit 61 KByte Speicherplatz herrschen. Neben dem für Normalsterbliche völlig überteuerten C-Compiler von Digital Research (jenseits der Schallmauer von 1000 Mark) tummeln sich in der Gegend um etwa 500 Mark mehrere Programme, zum Beispiel MI-C (auch auf 3-Zoll-Disketten lieferbar) oder auch BDS-C, dem selbst HiSoft in der Anleitung zum eigenen C-Compiler Lob zollt. Besonders interessant für C-Begeisterte mit kleinem Geldbeutel dürfte der Small C-Compiler sein, den Markt & Technik für 148 Mark herausgebracht hat (angepaßt für Schneider und Commodore).

Fast alle Compiler verlangen denselben C-Quellcode. Manche Eigenschaften, die von Ritchie und seinem Kollegen Brian W. Kernighan im C-Standardwerk »The C Programming Language« (»Programmieren in C«), festgelegt sind, wurden aber in diversen Compilern nicht implementiert. Oft fehlen Bitfelder und Fließkommazahlen. Um Sie nicht zu verwirren, gehen wir in diesem Kurs auf solche eher ausgefallenen Dinge nur am Rande ein.

Um C zu verstehen, sollten Sie die im Text angegebenen Programmbeispiele nach Möglichkeit nachvollziehen. Wie ein Programm compiliert und gestartet wird, das steht in Ihrem Handbuch. CP/M-Programme erwarten den C-Quellcode meist in Form einer Textdatei, die zuvor mit einem Texteditor erzeugt wurde. HiSoft-C verlangt zum Beispiel den Befehl »HC PROGRAM.C« und macht daraus ein COM-File mit dem Namen »PROGRAM.COM«. Andere CP/M-Compiler erzeugen oft nur einen Assembler-Quellcode, der dann mit einem Assembler, wie ASM.COM, MAC.COM oder MACRO-80, übersetzt wird.

Kernighan und Ritchie, im C-Insider-Jargon meist nur noch K&R genannt, bringen als erstes Beispiel ein Programm, das den Satz »Hello, world« auf dem Bildschirm ausgibt. Da praktisch jeder C-Kurs so beginnt, wollen wir uns dem Trend nicht verschließen – aber in Deutsch.

Etwas ganz Wichtiges, das Sie bei der Eingabe der Programme beachten müssen, sollte im voraus erwähnt werden. C-Programme werden mit Vorliebe in Kleinbuchstaben geschrieben. Während die Schreibweise den meisten Basic-Interpretern völlig egal ist, unterscheidet ein Großteil der C-Compiler aber genau zwischen Namen wie »main«, »MAIN« und »Main«.

Doch nun zu unserem ersten Programm. Falls Ihnen der Ausgabetext nicht gefällt, können Sie ihn natürlich durch einen anderen ersetzen:

```
main()
{
 printf("Hallo, Welt!\n");
}
```

Sollte eine Fehlermeldung erscheinen, obwohl Sie keinen Fehler finden können, so setzen Sie vor die erste Zeile den Befehl »#include "STDIO"« oder »"STDIO.H"«.

»STDIO« ist eine Diskettendatei. Vielleicht hat sie bei Ihrem Compiler einen anderen Namen. Lassen Sie sich am besten das Inhaltsverzeichnis der Compilerdiskette ausgeben.

Wenn Sie das Programm starten, erscheint auf dem Bildschirm:

```
Hallo, Welt!
```

Sie sind natürlich nicht gezwungen, das C-Programm sklavisch nach der Vorlage einzugeben, denn der Compiler erwartet kein festes Format. Ebensogut könnten Sie schreiben:

```
main() { printf ("Hallo, Welt!
\n");}
```

Oder auch:

```
                        main() {
printf("Hallo, Welt!\n");
                     }
```

Denken Sie aber an folgendes: Wenn Sie Ihre Programme nach einigen Monaten wieder anschauen, wollen Sie auch verstehen, was Sie programmiert haben. Zur logischen Gliederung eignen sich die Einrückungen hervorragend. Der Compiler versteht nämlich Programme in jedem Bildschirmformat – Sie auch?

Gehen wir der Reihe nach die Befehle durch. C-Programme bestehen ausschließlich aus »Funktionen«. Funktionen in C sind das, was man in Basic mit GOSUB oder FNx(y) aufruft und was in Pascal mit FUNCTION oder PROCEDURE bezeichnet wird. In C existiert also kein Unterschied zwischen Funktionen und Unterprogrammen.

Jedes C-Programm besteht aus mindestens einer Funktion. Die Hauptfunktion erhält den Namen »main« und wird beim Programmstart automatisch aufgerufen. »main« kann von sich aus andere Funktionen – sofern vorhanden – aufrufen. Dabei werden in Klammern Variablen als Argumente übergeben. Da das in unserem Fall nicht nötig ist, müssen Sie dem Funktionsnamen leere Klammern folgen lassen: »main()«

Der gesamte Programmcode, der hier lediglich die printf-Anweisung enthält, wird mit geschweiften Klammern ({ und }) umschlossen. Nach jedem Befehl steht ein Strichpunkt. Dieser trennt – wie in Pascal – die einzelnen Befehle voneinander.

printf hat die Aufgabe, Texte und Zahlen formatiert auszugeben. printf gehört nicht zur Sprache C, sondern ist eine Funktion. C selbst ist eine Sprache mit sehr geringem Umfang. Insbesondere fehlen Ein- und Ausgabefunktionen völlig. Die eigentlichen C-Befehle dienen vorrangig der Steuerung des Programmflusses. Die Funktion printf finden Sie hingegen bei Ihrem Compiler wahrscheinlich auf der Diskette in einer Bibliotheksdatei mit dem Namen »STDIO« oder »STDIO.H«, das steht für »Standard Input/Output«. Entsprechend der Normung der C-Sprache fallen auch überraschende Übereinstimmungen zwischen den C-Bibliotheken verschiedener Compiler auf. Sie orientieren sich fast alle an der Unix-Bibliothek von Kernighan und Ritchie.

printf gibt den String aus, der in Anführungszeichen steht. Das Zeichen »\« dient der Ausgabesteuerung. Nachfolgende Buchstaben werden zum Beispiel dazu verwendet, den nächsten Tabulatorstop zu suchen oder einen Zeilenvorschub auszulösen:

\b	Backspace (Cursor ein Zeichen nach links bewegen)
\f	Form Feed (Löschen des Bildschirms)
\n	Newline (Wagenrücklauf und Zeilenvorschub)
\r	Return (Wagenrücklauf)
\t	Tab (nächste Tabulatorposition anspringen)
\0	Nullbyte (String-Begrenzer)
\\	Das Zeichen »\« selbst
\'	Einfaches Anführungszeichen (Apostroph)
\"	Anführungszeichen (")

Statt dieser Kennbuchstaben dürfen Sie auch direkt Bytes im Oktalsystem angeben:

```
printf("\0 \1 \2 \3");
```

Die Werte zwischen oktal 0 (dezimal 0) und oktal 37 (dezimal 31) dienen bei den meisten Computern für Bildschirm-Steuerungsfunktionen, wie Farbenauswahl, Inversdarstellung, Bildschirmmodus und ähnliche Dinge. Da die Codes nicht genormt sind, können C-Programme, die auf diese Funktionen angewiesen sind, nicht ohne weiteres auf andere Computer übertragen werden.

Experimentieren Sie ruhig mit diesen Steuermöglichkeiten. Denn mit ihnen läßt sich schon einiges Interessante anstellen:

```
main()
 {
  printf("\f Zeile1\n Zeile2\t
         Tab1\t Tab2 \\");
}
```

Aber die Funktion printf kann noch viel mehr. Sie rechnet beispielsweise Zahlen in verschiedene Zahlensysteme um und druckt diese dann formatiert aus. Dazu geben Sie statt des Textes einen Format-String an, der zum Beispiel so aussieht:

```
main()
{
  printf("%x",32767);
}
```

Die Formatkenner werden durch ein Prozentzeichen eingeleitet. Danach folgt ein Buchstabe, der das Format angibt: %x besorgt beispielsweise die hexadezimale Ausgabe einer Zahl, statt 32767 wird hier also »7FFF« ausge-

druckt. Die wichtigsten Formatkenner lauten:

%d	Dezimale Ausgabe einer Integerzahl (decimal)
%o	Oktale Ausgabe einer Integerzahl (octal)
%u	Dezimale vorzeichenlose Darstellung (unsigned)
%x	Hexadezimale Ausgabe (hexadecimal)
%c	Ausgabe als ASCII-Zeichen (character)

Die Anzahl der Formatkenner muß mit der der angegebenen Zahlen übereinstimmen. Gibt es da Unterschiede, führt das meist zu recht unsinnigen Resultaten, die mitunter recht amüsant sind. C schreibt vor, daß alle Variablen, die in einem Programm benutzt werden, definiert sein müssen. Diese Definition erfolgt im Programmkopf. Die wichtigsten Datentypen in C sind:

int	Integerzahlen zwischen -32768 und 32767
char	Einzelne Buchstaben, (ASCII-Code von 0 bis 255)
float	Fließkommazahlen (Wertebereich vom Compiler abhängig)

Abwandlungen dieser Typen sind möglich. Zum Beispiel gibt es long int (32-Bit-Integerwerte – 0 bis 4294967295), short int (8-Bit-Integerzahlen - 0 bis 255), unsigned int (16-Bit-Integers ohne Vorzeichen, Wertebereich daher von 0 bis 65535) und double, das sind doppeltgenaue Fließkommazahlen. Allerdings sind die Bitbreiten und Wertebereiche nicht von Ritchie festgelegt worden. Sie werden meistens aus der Maschinensprache eines Prozessors direkt abgeleitet. So verarbeitet ein Großcomputer mit einem 32-Bit-Prozessor sicher auch »int« als 32-Bit-Integerzahlen.

Strings (Zeichenketten) sind in der C-Sprache nicht vorgesehen. Sie werden aus einzelnen Zeichen oder Feldern des Typs char zusammengestellt. Wertezuweisungen erfolgen auf die den Basic-Programmierern bekannte Art:

```
main()
{
 int x; /* Definition von x als
          Integer */
 x=64;  /* Zuweisung von 64 an
          x */
 printf("%d %x %o %c",x,x,x,x);
}
```

Kommentiert werden Programme wie oben angegeben: »/* Das ist ein Kommentar */«. Kommentare dürfen überall stehen, wo auch ein Leerzeichen oder ein Wagenrücklauf stehen könnte – nur nicht in Zeichenketten. Schachteln Sie auch keine Kommentare - das geht garantiert schief. Diese Einschränkungen sollten Sie aber nicht daran hindern, Ihre Programme möglichst ausführlich mit Kommentar-Zeilen zu erläutern. Wie bei allen Compilersprachen wirken sich Kommentare nicht auf die Länge oder die Ablaufgeschwindigkeit des erzeugten Programms aus.

»int x« bestimmt, daß die Variable x im Programm eine Integervariable ist. »x=64« weist der Variablen den Wert 64 zu. printf schließlich gibt x in verschiedenen Formaten aus: dezimal (64), hexadezimal (40), oktal (100) und als ASCII-Zeichen (@).

Statt einer Einzelvariablen können Sie auch mehrere auf einen Schlag angeben:

```
int   x,y,z;
char  c1,c2,c3,c4;
float a,b,g,h;
```

Mit den C-Variablen läßt sich genauso rechnen wie in Basic:

```
main()
 {
   int x,y;
   y=23+4*5;
   x=y*(2+y-3);
   y=x/2;
   printf("%d %d",x,y);
 }
```

Alle vier Grundrechenarten sind also vorhanden (Addition »+«, Subtraktion »–«, Multiplikation »*« und Division »/«). Leider ist die Hierarchie der Operatoren in C ziemlich kompliziert und auch nicht immer genau festgelegt. Wenn Sie also Zweifel an der Ablauffolge haben, setzen Sie sicherheitshalber Klammern. Das ist jederzeit erlaubt.

Da C eine sehr maschinennahe Sprache ist, entdecken Compiler und Laufzeitbibliothek (das ist der Code, der zu jedem übersetzten Programm dazugebunden wird, damit es ablaufen kann) normalerweise keine Über- oder Unterschreitungen des Wertebereichs. Die Programme schneiden einfach den Teil der Zahl weg, den sie nicht verwerten können. Wenn Sie zum Beispiel als Ergebnis einer ganzzahligen Rechnung mit Integervariablen eine 20 Bit breite Zahl erhalten, »vergißt« das Programm die vier vordersten Bits und läßt nur die sechzehn darstellbaren übrig.

Was wäre eine Programmiersprache wert, die jedes Programm nur von Anfang bis Ende abarbeiten könnte, ohne den Programmfluß zu steuern? Basic ist auf diesem Gebiet recht spartanisch ausgestattet, denn die meisten Interpreter bieten nur IF-THEN- und FOR-NEXT-Konstruktionen an – wenn es hochkommt vielleicht auch WHILE-WEND. Und natürlich das berühmtberüchtigte, eher verwirrende GOTO.

Mit IF kann man auch in C programmieren. Nur ist die Syntax der if-Anweisung (Kleinschreibung in C!) etwas anders:

```
if (Bedingung) Aktion;
```

Trifft die Bedingung, die in Klammern gesetzt ist, zu, wird der Befehl »Aktion« ausgeführt. Setzen Sie aber keinesfalls zwischen Bedingung und Aktion einen Strichpunkt. In diesem Fall interpretiert der Compiler den Strichpunkt als Leerbefehl. Die Aktion wäre dann unabhängig von der if-Bedingung und würde immer ausgeführt werden.

Ein Programmbeispiel:

```
main()
 {
 int c;
 c=getchar();
 if (c==64) printf (" Klammeraffe
                      !\n");
            printf (" Der ASCII-
            Code ist %d",c);
 }
```

Die Funktion getchar() stammt aus der Bibliothek und liest ein Zeichen von der Tastatur ein. Sie übergibt der Variablen c den ASCII-Code des Zeichens. Wenn Sie den »Klammeraffen«, das ist das Zeichen »@«, eingegeben haben, meldet der Computer »Klammeraffe!«, und gibt zusätzlich dessen ASCII-Code aus.

Der Vergleich »c==64« ist keineswegs ein Druckfehler. Damit der Compiler ihn von einer Wertzuweisung »c=64« unterscheiden kann, müssen beim Vergleich zwei Gleichheitszeichen angegeben werden.

Hier alle Vergleichsoperatoren:

==	x==y	x ist gleich y
<	x<y	x ist kleiner als y
<=	x<=y	x ist kleiner oder gleich y
>	x>y	x ist größer als y
>=	x>=y	x ist größer oder gleich y
!=	x!=y	x ist ungleich y

Etwas ungewohnt dürfte die Schreibweise »!=« sein. Eine Eselsbrücke: Stellen Sie sich die beiden Symbole übereinander gedruckt vor. Dann erhalten Sie ein Zeichen ähnlich dem mathematischen für die Ungleichheit (»≠«).

Was machen Sie aber, wenn Sie nach der if-Bedingung nicht nur einen einzelnen Befehl, sondern eine ganze Befehlsgruppe ausführen lassen wollen? Sie erklären einfach die Befehle zu einem Block und umgeben diese Verbundanweisung mit geschweiften Klammern. Das entspricht exakt der BEGIN-END-Schachtelung in Pascal.

```
main()
 {
   int c;
   c=getchar();
   if (c != '*') {
               printf("Dies ist
               kein Sternchen
               !\n");
               printf("Es ist
               ein %c",c);
               }
}
```

Wenn Sie ein Zeichen eingeben, das kein Stern ist (ASCII-Code 42), werden die beiden printf-Funktionen ausgeführt. Andernfalls beendet der Computer das Programm. In der if-Zeile sehen Sie, daß auch String-Konstanten als Vergleichsobjekte dienen können:

```
if (c != 42)
if (c != '*')
if (c != 0x2A)
```

sind alle identisch.

Hexadezimale Zahlen müssen mit »0x« beginnen – zum Beispiel »0xFF«, »0x3C« oder »0x3FAC«.

Die if-Anweisung läßt sich um einen else-Teil erweitern, der ausgeführt wird, wenn die gegebene Bedingung nicht zutrifft:

```
main()
 {
   int c;
   c=getchar();
   if (c != 0x2A) printf
                 ("Kein Stern!");
                  else printf
                 ("Ein Stern!");
 }
```

Auch hier ist eine Verbundanweisung zulässig, die die Ausführung mehrerer von if oder else abhängiger Befehle erlaubt:

```
main()
 {
  int c;
  c=getchar();
  if (c != 42) { printf("Kein
                 Stern!$\n");
                 printf("Statt
                 dessen ein %c",
                 c);
               }
          else { printf("Ein
                 Stern!$\n");
                 printf("Der
                 ASCII-Code ist
                 42");
               }
  }
```

Wenn Sie eine Vorliebe für verwirrende Programme haben, können Sie auch if-Kommandos ineinander verschachteln. Wie wäre es mit folgendem Programm?

```
main()
 {
   int c;
   c=getchar();
   if (c >= 42) { if (c == 42)
         printf("ASCII-Code=42");
    else printf("ASCII-Code>42");
                 }
    else printf("ASCII-Code<42");
 }
```

Alles klar? Gerade dieses Programm läßt sich erheblich verständlicher mit drei gleichrangigen if's schreiben:

```
main()
 {
   int c;
   c=getchar();
   if (c>42) printf
             ("ASCII-Code>42);
   if (c==42) printf
             ("ASCII-Code=42);
   if (c<42) printf
             ("ASCII-Code<42);
 }
```

Für den Fall, daß ein Befehl oder eine Befehlsgruppe von mehreren Bedingungen gleichzeitig abhängt, könnten Sie – wie oben gezeigt – mehrere if's zusammenbasteln. Wesentlich übersichtlicher wird es aber mit logischen Operatoren, die den bekannten Funktionen AND, OR und NOT aus Basic entsprechen:

&& Logisches Und (AND)
|| Logisches Oder (OR)
! Logisches Nicht (NOT)

Demonstrieren lassen sich die Operatoren mit einem Programm, das drei Zeichen von der Tastatur entgegennimmt und dann vergleicht:

```
main()
 {
  int x,y,z; /* Speichern die
                3 Zeichen */
  x=getchar(); getchar();
  y=getchar(); getchar();
  z=getchar(); getchar();
  if (x=='A' && y=='B' && z=='C')
        printf("Alphabet!\n");
        /* 1 */
  if (x>='1' && x<='9')
        printf("x ist Ziffer\n");
        /* 2 */
  if (x=='+' || y=='+')
        printf("x oder y = +");
        /* 3 */
 }
```

Im Fall /*1*/ gibt der Computer den Text »Alphabet!« aus, wenn Sie A, B und C eingeben. Fall /*2*/ meldet »Ziffer!«, wenn das erste Zeichen Ihres Inputs (entspricht der Variablen x) eine Ziffer ist. Der Fall /*3*/ zeigt das logische Oder. Ist das erste oder das zweite Zeichen ein Plus »+«, schreibt das Programm die Meldung »x oder y = +« auf den Bildschirm.

Die Gliederung

```
x=getchar(); getchar();
y=getchar(); getchar();
z=getchar(); getchar();
```

ist notwendig, weil die getchar-Funktion auch den Code der Return- oder Enter-Taste (10 oder 13) registriert. Um diesen jeweils auszufiltern, muß ein »Dummy«-getchar eingesetzt werden. An diesem Beispiel ist recht gut zu erkennen, daß C-Funktionen entweder keinen oder genau einen Parameter ans aufrufende Programm übergeben können. x=getchar() fragt die Tastatur ab und liefert den Code der gedrückten Taste in der Variablen x ab. getchar() macht das gleiche, doch der Tastenwert geht verloren. Eine solche Funktion, die keinen Wert zurückgibt, ist eigentlich nichts anderes als ein Unterprogramm oder eine Prozedur in einer der herkömmlichen Programmiersprachen.

Neben diesen logischen Operatoren befinden sich noch zahlreiche weitere im C-Sprachumfang:

– Der Minus-Operator: -a=0-a. Dieser Operator zeigt an, daß eine Zahl negativ ist, genau wie in Basic (beispielsweise »B=-A« oder »Z=-3*A«). Seltsamerweise ist in C kein Plus vor Variablen und Zahlen erlaubt. x=+a ist also verboten – aber eigentlich auch unsinnig. Hingegen ist x=b+a eine Selbstverständlichkeit. So können Sie statt x=+a auch x=0+a schreiben. Allzu viel Sinn steckt aber nicht hinter solch einer Befehlsfolge.

– Das Einerkomplement: ~a dreht alle Bits in der angegebenen Variablen oder Konstanten um. Aus Nullen werden Einsen und aus Einsen Nullen. Damit ist ~255=-256.

– Die Inkrementoperatoren: a++ und ++a. Beide Operatoren erhöhen den Wert von a um Eins. Der Unterschied liegt im Zeitpunkt, zu dem das geschieht:

`a=3; printf("%d",++a);`

druckt "4", a ist 4.

++a ist ein Präfix-Operator.

`a=3; printf("%d",a++);`

druckt "3", a ist 4.

a++ ist ein Postfix-Operator.

Mit den Inkrement-Anweisungen ist eine Zählschleife sehr leicht zu konstruieren:

```
main()
 {
   int i;
   i=0;
   while (i>1000) printf
                  ("%d\n",i++);
 }
```

Ausführlich gehen wir auf while weiter unten ein.

– Die Dekrementoperatoren: a-- und --a.

Sie vermindern den Wert von a um Eins. Es gelten dieselben Regeln wie bei den Inkrementbefehlen.

– Die Shift-Befehle: Sie schieben die Bits in einer Variablen um einen bestimmten Wert nach links oder rechts.

`a=2; a<<1`

ist der Links-Shift. a hat den Wert 4 (2*2).

`a=8; a>>2`

ist der Rechts-Shift. a hat den Wert 2.

– Die bitweisen Logikoperatoren: Die oben erklärten Logikoperatoren && und \\ liefern immer nur die Werte 0 und 1, je nachdem, wie die beiden Operanden aussehen. Die bitweisen Logikoperatoren verknüpfen die Operanden Bit für Bit und weisen dann der angegebenen Variablen das Ergebnis zu.

a&b ist das bitweise logische Und. 127 & 255 ergibt 127 (0111 1111 bin AND 1111 1111 bin = 0111 1111 bin).

a\b ist das bitweise logische Oder. 128 \ 3 ist 131 (1000 0000 bin OR 0000 0011 bin = 1000 0011 bin).

a↑b hat in C absolut nichts mit der Potenzfunktion zu tun, sondern steht für das bitweise logische Exklusiv-Oder. 1 ↑ 1 ist 0 (01 bin XOR 01 bin = 00 bin). (Martin Kotulla/hg)

Erste Schleife, zweite Schleife – C

C kennt verschiedene Wege, Schleifen zu konstruieren. Um sie alle zu zeigen, geben wir in unseren Beispielsprogrammen den gesamten ASCII-Zeichensatz auf dem Bildschirm aus – und zwar in folgender Form:

DEZ	HEX	OKT	ASCII
32	20	40	
33	21	41	!
34	22	42	"
.	.	.	
.	.	.	
255	FF	377	

In Basic wäre wohl die einfachste Art, so etwas zu programmieren, die:

```
100 PRINT "DEZ HEX OKT ASCII"
110 FOR I=32 TO 255
120 PRINT I;HEX$(I);OCT$(I);
    CHR$(I)
130 NEXT I
```

Voraussetzung ist natürlich, daß der Basic-Interpreter die Funktionen HEX$ und OCT$ versteht. HEX$ ist ja noch relativ geläufig, aber Oktalzahlen ?

C bietet die Zahlenkonversionen grundsätzlich mit der printf-Funktion. Schwierigkeiten bereitet uns aber noch die Übersetzung der FOR-Schleife in C. »for« ist gleichzeitig auch der Befehlsname in C. Nur der Aufbau ist anders als in Basic:

```
for (Ausgangsbedingung;
    Endbedingung; Zählbedingung)
```

Eine Schleife, die von 0 bis 10000 zählt und diese Zahlen ausgibt, gibt man in C so ein:

```
for (i=0; i<=10000; i=i+1)
    printf("%d ",i);
```

Viel schwerer als in Basic gestaltet sich die Ausgabe des Zeichensatzes auch nicht.

Das Programm sieht so aus:

```
main()
{
    int i;
    printf("DEZ HEX OKT
          ASCII\n\n');
    for (i=32; i<=255; i=i+1)
        printf("%d %x %o %c
                \n",i,i,i,i);
}
```

i=32 ist die Startbedingung. Der Variablen i wird hier der Wert 32 zugewiesen. Dann führt der Computer den printf-Befehl aus. Er kehrt zur for-Anweisung zurück und prüft, ob die zweite Bedingung i<=255 noch zutrifft. Ist sie nicht mehr erfüllt, beendet das Programm die Schleife. Sonst gilt i=i+1, und die Schleife wird erneut abgearbeitet.

Wiederholung – das ist eins der Zauberworte der Computer. C kennt natürlich auch Schleifen und andere Programmstrukturen, die bestimmte Befehlsfolgen immer wieder aufrufen.

Der besondere Aufbau der for-Schleife schreibt nicht zwingend vor, bei allen drei Bedingungsteilen dieselbe Variable zu verwenden. Sie können sogar einzelne Teile völlig weglassen. Ein Programm, das eine Zeichenkette von der Tastatur liest und in der Folge auf dem Bildschirm ausgibt, kann damit so aussehen:

```
main()
{
    int c;
    c=0;
    for ( ; c != 10 ; )
        {c=getchar();
            printf("%c",c);
        }
}
```

Die Schleife druckt solange Zeichen aus, bis sie ein Line-Feed (ASCII-Code 10) entdeckt. Sie ist übrigens ein hervorragendes Beispiel dafür, daß ein Programm – trotz nahezu vollständiger Kompatibilität der Sprache – auf unterschiedlichen Computern anders reagiert. Manche Computer puffern die Eingabe, das heißt, sie legen die gelesenen Zeichen erst intern im Speicher ab, bis der Benutzer die Zeilenschaltung (ENTER oder RETURN) betätigt. Andere Geräte verwenden die unmittelbare Ein- und Ausgabe. Je nachdem, welchen Computer und welches Betriebssystem Sie benutzen, erhalten Sie für den Satz »Dies ist ein C-Text«:

Dies ist ein C-Text Dies ist ein C-Text

oder

DDiieess iisstt eeiinn CC-TTeexxtt

In vielen Fällen ist aber eine for-Schleife nicht der Weisheit letzter Schluß. Deshalb bietet C zwei weitere Konstruktionen an: »while« und »do-while«. Beginnen wir mit »while«. Dieser Befehl führt die nachfolgenden Anweisungen oder die folgende Befehlsgruppe aus, solange eine Bedingung erfüllt ist:

```
while (Bedingung) Aktion;
```

Das Zeichensatz-Programm läßt sich mit while umschreiben – auch wenn es dadurch unübersichtlicher wird:

```
main()
{
    int i;
    i=32;
    while (i<=255)
      {
        putchar(i);
        i=i+1;
          }
}
```

Damit der Compiler weiß, was alles abhängig von while ausgeführt werden soll, müssen Sie die Befehle zu einer Verbundanweisung zusammenfassen und mit »{« und »}« umklammern.

Neu ist hier die Funktion putchar. Sie gibt ein einzelnes Zeichen vom Typ Integer oder Char auf dem Bildschirm aus. Gleichwertig dazu steht »printf("%c", i)«. printf erzeugt zwar einen längeren Programmcode, dafür ist diese Funktion aber auch universeller. Allerdings vermag putchar durchaus auch \-Formatkenner (\f zum Löschen des Bildschirms, \t für Tabulatorsprung und so weiter) zu verarbeiten. Ein putchar('\f') ist – von der Programmlänge her gesehen – einem printf("\f") vorzuziehen.

while prüft vor der Ausführung der Schleife, ob die angegebene Bedingung zutrifft. Das kann auch bedeuten, daß die Schleife unter Umständen überhaupt nicht bearbeitet wird:

```
main()
{
    while (3>4) putchar(64);
}
```

Da 3 natürlich niemals größer als 4 ist, wird kein einziger Klammeraffe (ASCII-Code 64) ausgegeben.

Anders arbeitet do-while:

```
do
   Aktion;
 while (Bedingung);
```

Hier prüft der Computer erst nach mindestens einmaliger Ausführung der Aktion, ob die Bedingung wahr ist.

```
main()
  {
    do
      putchar(64);
    while (3>4);
  }
```

Natürlich benötigt do-while keine Klammerung der abhängigen Anweisungen, da diese do und while schon umschließen. So sind die Befehle für den Compiler als Verbundanweisung erkennbar, und es können gar keine Verständniskonflikte auftreten.

Manchmal tritt der Fall ein, daß der Computer eine Schleife vor dem eigentlichen Ende beenden soll, zum Beispiel wenn ein Ergebnis einer Berechnung bereits vorliegt oder ein bevorstehender Fehler abgefangen werden soll. Dafür kennt C den break-Befehl. Ihn demonstriert ein Programm, das eine Eingabe von der Tastatur entgegennimmt und die ASCII-Codes aller Zeichen addiert. Abgebrochen werden soll das Programm, wenn ein Sternchen (ASCII-Zeichen 42) entdeckt wird:

```
main()
  {
    int c,summe;
    summe=0; /* Mit Anfangswert
                 initialisieren */
    while (1) /* Endlosschleife */
     {
       c=getchar();
       if (c==42) break;
       summe=summe+c;
     }
    printf("Summe: %d",summe);
  }
```

Das Programm definiert sich die zwei Variablen c und summe. c empfängt jeweils ein Zeichen von der Tastatur, und summe addiert die ASCII-Codes. »while(1)« ist ein beliebter Trick, um den Computer in einer Endlosschleife festzuhalten. Da die Bedingung immer erfüllt ist, beendet das Programm die Schleife nie aufgrund dieser while-Anweisung. Also muß irgendeine andere Abbruchbedingung gefunden werden, in unserem Beispiel »if (c==42) break;«. Nachdem der Computer ein Zeichen von der Tastatur gelesen hat (c=getchar();), prüft er nach diesem Befehl, ob es sich um ein Sternchen handelt. break befiehlt dem Computer, die while-Schleife zu verlassen und als nächstes den ersten Befehl hinter der Schleife abzuarbeiten. Das ist in diesem Programm eine printf-Funktion, die die Summe der ASCII-Codes ausgibt.

Ebenso läßt sich break auch in for- und do-while-Schleifen verwenden. Auch dort bewirkt der Befehl einen Abbruch der Schleifenausführung:

```
main()
  {
    int i;
    for (i=0; i<1000; i=i+1)
      {
        printf("%d\n",i);
        if (i==500) break;
      }
    printf("Ende!");
  }
```

Die for-Anweisung schreibt dem Computer zwar vor, von 0 bis 1000 zu zählen und die einzelnen Werte auszugeben, doch er bricht aufgrund der Zeile »if (i==500) break« die Zählung bei 500 ab.

Den gegenteiligen Effekt von break bewirkt continue: continue ruft diejenige Programmzeile auf, in der die Schleifenbedingung getestet wird, und führt damit die restlichen Kommandos, die eventuell noch in der Schleife folgen, nicht aus. Der Befehl bezieht sich immer auf die innerste Schleife. Wenn mehrere Schleifen ineinander geschachtelt sind, kann continue also nicht benutzt werden, um eine äußere Schleife aufzurufen.

Ein Beispiel für continue in einer for-Schleife:

```
main()
 {
  int i;
  for (i=0; i<100; i=i+1)
   {
     if ((i % 10) != 0) continue;
     printf("%d\n",i);
   }
 }
```

Dieses C-Programm gibt nacheinander die Zahlen 10, 20, 30, 40 und so weiter bis 90 aus: es zählt in der for-Schleife von 1 bis 100 in Einerschritten hoch. In der if-Zeile werden aber alle Zahlen, die keine Zehner sind, ausgefiltert. Dazu bietet C einen Modulo-Operator, das Prozentzeichen. Dieser Rechenoperator gibt den Rest einer ganzzahligen Division an:

20/3 = 6,	6*3 = 18,	20-18 = 2	Rest: 20 % 3 = 2
40/5 = 8,	8*5 = 40,	40-40 = 0	Rest: 40 % 5 = 0
191/3 = 63,	63*3 = 189,	191-189 = 2	Rest: 191 % 3 = 2

In Pascal und in manchen Basic-Dialekten heißt dieser Operator »mod«:

```
100 INPUT a,b
110 PRINT a\b,a MOD b
120 GOTO 100
```

Läßt sich eine Zahl nicht ohne Restbetrag durch 10 teilen, ist sie keine Zehnerzahl. Sie wird folglich von continue »verschluckt«. Die übrigen Zahlen gibt die printf-Funktion auf dem Bildschirm aus.

Neben if gibt es in C noch eine weitere Möglichkeit, Vergleiche anzustellen. Dieser Befehl heißt »switch-case« und ist immer dann zu empfehlen, wenn viele Vergleiche mit einer einzigen Variablen durchgeführt werden sollen. Ein Programm, das ein Symbol von der Tastatur einliest und dann untersucht, schreiben Sie so:

```
main()
 {
   int c;
   c=getchar();
   if (c==32) printf
      ("Ein Leerzeichen!\n");
   if (c==33) printf
      ("Ein Ausrufezeichen!\n");
   if (c==34) printf
      ("Ein Anführungs-
      zeichen!\n");
   if (c==35) printf
      ("Ein Doppelkreuz!\n");
   if (c==36) printf
      ("Ein Dollarsymbol!\n");
   if (c==37) printf
      ("Ein Prozentzeichen!\n");
 }
```

Spätestens nach dem zehnten if mag Ihnen das alles zu umständlich vorkommen. Es ist auch nicht die eleganteste Art zu programmieren. So geht es einfacher:

```
main()
  {
  int c;
  c=getchar();
  switch(c)
  {
  /*1*/ case 32: printf
         ("Ein Leerzeichen!\n");
  /*2*/ case 33: printf
         ("Ein Ausrufezeichen!\");
  /*3*/ case 34: printf
         ("Ein Anführungs-
         zeichen!\");
  /*4*/ case 35: printf
         ("Ein Doppelkreuz!\");
  /*5*/ case 36: printf
         ("Ein Dollarsymbol!\");
  /*6*/ case 37: printf
         ("Ein Prozentzeichen!\");
     }
}
```

Bis auf den switch-Teil mit der Verbundanweisung ist das Programm identisch mit dem vorigen. In switch wird die Variable angegeben, die zu überprüfen ist, hier die Variable c. In einer, durch geschweifte Klammern umgebenen, Verbundanweisung stehen nach case die Fälle, von denen ausgehend Befehle ausgeführt werden. Die »/*Kommentare*/« sind nur angegeben, um einen Bezug auf die Zeilen herzustellen. Die Arbeit des Computers veranschaulichen Sie sich so:

```
int c;       Ich definiere eine Variable c als Integer
switch(c)    Aha, jetzt soll ich c untersuchen
case 32:     Hat c den Wert 32?
             Ja - dann muß ich »Ein Leerzeichen!« drucken
             Nein - dann unternehme ich nichts
case 33:     Hat c den Wert 33?
             Ja - ich gebe »Ein Ausrufezeichen!« aus
             Nein - ich mache gar nichts
...
case 37:     Ist c=37?
             Ja - ich melde »Ein Prozentzeichen!«
             Nein - ich habe ja überhaupt nichts zu tun!
```

Wenn Sie das Programm mit Ihrem C-Compiler übersetzen und laufen lassen, geben Sie zuerst einmal das Prozentzeichen (%) ein. Der Computer meldet: "Ein Prozentzeichen!"

Das Programm arbeitet aber noch nicht korrekt. Denn tippen Sie statt des Prozentsymbols ein Ausrufezeichen ein, so gibt es einiges an Text aus:

```
Ein Ausrufezeichen!
Ein Anführungszeichen!
Ein Doppelkreuz!
Ein Dollarsymbol!
Ein Prozentzeichen!
```

Die Lösung für diesen »Fehler« ist recht simpel: Der case-Befehl versteht sich nämlich nicht wie Pascals CASE, sondern eher wie der GOTO-Befehl in diesem Basic-Programm:

```
10 INPUT a$
20 ON ASC(a$)-32 GOTO 32,33,34,35,
   36,37
32 PRINT "Ein Leerzeichen!"
33 PRINT "Ein Ausrufezeichen!"
34 PRINT "Ein Anführungszeichen!"
35 PRINT "Ein Doppelkreuz!"
36 PRINT "Ein Dollarsymbol!"
37 PRINT "Ein Prozentzeichen!"
```

case durchläuft also alle folgenden Fälle und führt sie ohne Prüfung der Bedingung aus. Da dies meistens unerwünscht ist, ist Abhilfe bereits vorgesehen: Der break-Befehl, den wir vorhin im Zusammenhang mit Schleifenkonstruktionen erwähnt haben, verläßt ebenso eine switch-case-Anweisung:

```
main()
{
  int c;
  printf("Bitte die Nummer
          eines Wochentags: ");
  c=getchar()-48;
  switch(c)
   {
     case 1: printf
             ("Montag"); break;
     case 2: printf
             ("Dienstag"); break;
     case 3: printf
             ("Mittwoch"); break;
     case 4: printf
             ("Donnerstag");
             break;
     case 5: printf
             ("Freitag"); break;
     case 6: printf
             ("Samstag"); break;
     case 7: printf
             ("Sonntag"); break;
   }
  printf(" ist der %d. Wochen-
         tag.",c);
}
```

Wenn Sie hier eine Zahl zwischen 1 und 7 eingeben, antwortet der Computer mit dem korrespondierenden Wochentag:

```
Bitte die Nummer eines Wochentags: 1
Montag ist der 1. Wochentag.
Bitte die Nummer eines Wochentags: 4
Donnerstag ist der 4. Wochentag.
Bitte die Nummer eines Wochentags: 7
Sonntag ist der 7. Wochentag.
```

Entfernen Sie übungsweise einmal überall die break-Anweisungen. Sie sehen dann, wie der Computer alle Wochentage »durchrutschen« läßt.

Falls Sie ein Symbol eingegeben haben, das keine Ziffer oder aber die Zahlen »8« oder »9« darstellt, ignoriert der Computer einfach die case-Anweisungen und macht nach der Verbundanweisung – hier also bei »printf« – weiter, als wäre nichts geschehen. Oft ist aber eine Fehlermeldung angebracht. Der Befehl, der es erleichtert, solche Programm- oder Eingabefehler abzufangen, heißt »default«:

```
main()
 {
  int i;
  printf("Wählen Sie:A oder B");
  i=getchar();
  switch(i)
   {
     case 'A':  printf
     ("Ein A also!");
     break;
     case 'B':  printf
     ("Sie haben B getippt!");
     break;
     default:   printf
     ("Eingabefehler! ");
     printf
     ("Weder A noch B.");
   }
 }
```

Das Programm fordert von der Tastatur einen Buchstaben an, wahlweise »A« oder »B«. Für diese beiden Symbole ist das Verhalten genau definiert. Der Computer führt die zugehörige case-Anweisung aus. Bei anderen Eingaben ruft er »default« auf.

Auch die Ausführung eines einzelnen Befehls auf mehrere case-Anweisungen hin ist zulässig:

```
switch(x)
case 1:
case 2: printf("1 oder 2");
```

Das »Wochentags-Programm« kann man so zu einem »Wochenende-Programm« machen:

```
main()
 {
   int c;
   printf("Bitte die Nummer
          des Wochentags: ");
   c=getchar()-48;
   switch(c)
    {
      case 1:
      case 2:
      case 3:
      case 4:
      case 5: printf
      ("Wochentag"); break;
      case 6:
      case 7: printf
      ("Endlich Wochenende!");
      break;
    }
 }
```

Die Fälle 1 bis 5 werden mit Wochentag beantwortet. Entdeckt der Computer aber eine 6 oder 7, meldet er freudig »Endlich Wochenende!«

Man glaubt es kaum, welchen Horror ein kleines Wörtchen auf viele Programmierer ausübt. Struktur-Fanatiker scheuen den Befehl »GOTO« in etwa demselben Maße wie Vampire den herabrechenden Morgen. Es stimmt sicher, daß die allzu häufige Anwendung von GOTO nicht gerade zu übersichtlichen Programmen führt. Aber in manchen Fällen ist dieser unbedingte Sprung in allen Programmiersprachen sehr nützlich. Letztlich gibt es GOTO sogar in vielen Pascal-Versionen. C steht da nicht zurück:

```
goto sprungziel;
 ...
 sprungziel: ...
```

Das folgende Beispiel eignet sich bestens, um zu zeigen, daß sich GOTO aber fast immer vermeiden läßt:

```
main()
   {
    int i;
    i=32;
    sprungziel: printf("%d = %x
                = %o = %c{n"
                ,i,i,i,i);
                i=i+1;
                if (i<=255)
                goto sprungziel;
   }
```

Das Programm gibt, wie unschwer zu erkennen ist, den ASCII-Zeichensatz und die verschiedenen Zahlenäquivalente in dezimaler, hexadezimaler und oktaler Notation aus. Erinnern Sie sich noch, wie elegant im Gegensatz dazu die Verwendung der for-Schleife wirkt?

Bisher verwendeten Sie nur die Funktion main(), die das eigentliche Hauptprogramm darstellt. Ebenso einfach können Sie Unterfunktionen definieren. Ob Sie diese vor oder hinter main() ablegen, ist dem Compiler völlig egal.

```
main()
   {
    printf
    ("Ich bin in main()\n");
    subfunction(); /* Aufruf
    der Funktion */
    printf
    ("Ich bin wieder main()\n");
   }

 subfunction()
   {
    printf
    ("Ich bin subfunction()\n");
   }
```

Genauso gut können Sie subfunction() vor main() angeben.

```
subfunction()
   {
    printf
    ("Ich bin subfunction()\n");
   }
main()
   {
    printf
    ("Ich bin main()\n");
    subfunction(); /* Aufruf der
    Funktion */
    printf("Ich bin
    wieder main()\n");
   }
```

Beim Programmstart wird main() aufgerufen und gibt den Text »Ich bin in main()« aus. Dann findet der Computer



den Aufruf der Unterfunktion subfunction() und bearbeitet diese. Dort steht der Ausgabebefehl »Ich bin subfunction()«. Da diese Funktion gleich wieder zu Ende ist, kehrt das C-Programm nach main() zurück. Es führt den nachfolgenden printf-Befehl aus: »Ich bin wieder main()«.

Die Bildschirmausgabe sieht also insgesamt so aus:

```
Ich bin main()
Ich bin subfunction()
Ich bin wieder main()
```

Am Ende des Funktionskopfes, also dort, wo der Namen der Funktion steht, dürfen Sie keinesfalls einen Strichpunkt setzen. Denn der Compiler verstünde dies als Funktionsaufruf anstatt als Definition.

Funktionsaufrufe lassen sich beliebig verschachteln, wenn Sie es nicht übertreiben und durch mehrere hundert Funktionen den gesamten Computer blockieren. Somit sind auch rekursive Programmiertechniken erlaubt.

Funktionen in C können aber nicht ihre »privaten« Unterfunktionen haben, wie das in Pascal erlaubt ist. In C sind alle Funktionen gleichwertig. Es gibt also keine von einer anderen Funktion abhängige Funktion, die nicht auch von einer dritten, unabhängigen Funktion aufgerufen werden kann. Diese Beschränkung gegenüber Pascal trägt sehr zur Verständlichkeit und Lesbarkeit von C-Programmen bei.

Sie dürfen also nicht so programmieren:

```
Funktionsdefinition 1
Funktionsdefinition 2
        Funktionsdefinition 2a
        Funktionsdefinition 2b
             Funktionsdefinition 2ba
             Funktionsdefinition 2bb
Funktionsdefinition 3
```

Statt dessen müssen Sie in C schreiben:

```
Funktionsdefinition 1
Funktionsdefinition 2
Funktionsdefinition 2a
Funktionsdefinition 2b
Funktionsdefinition 2ba
Funktionsdefinition 2bb
Funktionsdefinition 3
```

Während der erste Aufbau (Pascal) den Aufruf von 2bb aus 3 verbietet, ist dieser beim zweiten Schema, das den Syntax-Verelnbarungen von C entspricht, erlaubt.

Die Namen der Funktionen müssen – übrigens ebenso wie die Variablennamen – gewissen Konventionen entsprechen. So muß das erste Zeichen ein Buchstabe zwischen a und z beziehungsweise A und Z oder ein Unterstrich (_) sein. Die restlichen Zeichen sind Buchstaben, Ziffern oder Unterstriche. Von der Verwendung des Unterstrichs als erstem Symbol eines Namens ist aber abzuraten. Nach internationaler Gepflogenheit unter den C-Programmierern sind diese Namen nämlich für die interne Verwaltung des Compilers, zum Beispiel in den mitgelieferten Bibliotheksdateien, reserviert. Ein Verstoß gegen diese Regel kann Namenskollisionen zur Folge haben. Die maximal erlaubte Namenslänge hängt vom Compiler ab und wurde im C-Standard von Kernighan und Richie nicht exakt festgelegt. Bei den meisten Compilern dürfen Namen beliebig lang sein, es werden aber nur die ersten sechs oder acht Stellen unterschieden. Bei solchen Compilern sind damit die Namen »ein_langer_funktionsname1()« und »ein_langer_funktionsname2()« identisch – also aufgepaßt! Natürlich dürfen Namen einer Funktion oder Variablen nicht dem Namen eines Befehls (for, while, case und so weiter) entsprechen.

Damit sollte Ihnen der einfache Unterprogrammaufruf ohne Datenübergabe klar sein. Wichtig ist nur noch, anzumerken, daß jede Funktion – so auch main() – lokale Variablen verwendet. »Lokal« bedeutet, daß die Variablen einzig und allein für diese Funktion definiert sind. Andere Funktionen können auf sie nicht zugreifen:

```
main()
  {
    int x;
    x=3;
    printf("x ist in main()
           %d\n",x);
    subfunct();
    printf("x ist in main()
           immer noch %d\n",x);
  }
 subfunct()
  {
    int x;
    x=10;
    printf("x ist in subfunct()
           %d\n",x);
    x=x*2;
    printf("x ist jetzt in
           subfunct() %d\n",x);
  }
```

Die Variable x, die in main() definiert wurde, hat überhaupt nichts mit der Variablen x in subfunct() zu tun. Ihre einzige Übereinstimmung bleibt der Name. Keine Manipulation des Wertes in main() hat irgendwelche Auswirkungen auf den Wert in subfunct(). Dementsprechend liefert das Programm die folgenden Ausgaben:

```
main(): x=3
Meldung: »x ist in main() 3«
subfunct(): x=10
Meldung: »x ist in subfunct() 10«
subfunct(): x=x*2
Meldung: »x ist in subfunct() 20«
main():
Meldung: »x ist in main() immer noch 3«
```

Sie können aber auch sogenannte globale Variablen vereinbaren. Im Gegensatz zu den lokalen Variablen, deren Wirkungsbereich sich auf eine Funktion beschränkt, gelten die globalen in allen Funktionen. Sie werden vom Compiler als allgemeingültig erkannt, indem sie an den Anfang des Programms außerhalb der Funktionen aufgeführt werden:

```
int i;
main()
  {
    i=999; /* Wertzuweisung
              in main() */
    printf("In main() hat i
              den Wert %d\n",i);
    subfunct();
  {
 subfunct()
  {
    printf("Auch in subfunct()
              ist i=%d\n",i);
  }
```

Die Variable i steht in der ersten Zeile, also noch bevor main() oder subfunct() beginnen. Somit ist i global. In main() wird der Variablen der Wert 999 zugewiesen. Daß die Zuweisung korrekt ablief, beweist die printf-Zeile: »In main() hat i den Wert 999«. Der Befehl »subfunct();« überträgt die Kontrolle an die gleichnamige Funktion. Der Aufruf von printf bezeugt, daß i den Wert 999 behalten hat. Jede Veränderung des Werts von i in einer Funktion hat die Wertveränderung in allen anderen Funktionen zur Folge – ganz einfach, weil es sich immer um dieselbe Variable handelt.

In C haben lokale Variablen eine Vorrangstellung vor den globalen. Wenn Sie eine globale Variable definiert haben, können Sie ohne Probleme eine lokale Variable gleichen Namens initialisieren. Während der Laufzeit der zugehörigen Funktion wird die lokale Variable verwendet. Nach Ende der Bearbeitung benutzt der Computer wieder die globale Variable:

```
int i=123; /* globales i
              ist 123 */
main()
  {
    printf("Globales i ist
            %d\n",i);
    subfunct();
    printf("Globales i ist
            immer noch %d\n",i);
  }
subfunct()
  {
    int i; /* lokale Definition
              von i */
    i=999; /* lokale Wertzu-
              weisung */
    printf("Lokales i ist
            %d\n",i);
  }
```

Das C-Programm meldet:

```
Globales i ist 123.
Lokales i ist 999.
Globales i ist immer noch 123.
```

(Martin Kotulla/hg)

Daten werden unter C in verschiedenen Klassen gespeichert. Ob die Variable automatisch gelöscht wird oder im Speicher stehenbleibt – das muß der Programmierer beachten.

Speicherklassengesellschaft in C

C besitzt storage-classes, was in der deutschen Übersetzung »Speicherklassen« heißt. Dieser Ausdruck stammt aus der »C-Bibel« von Kernigham und Ritchie und wird hier näher beschrieben.

Es gibt drei Speicherklassen in C, nämlich »static«, »auto« und »register«. Normale Variablen, die bei ihrer Definition nicht anders bezeichnet werden, zählen zum »automatic«-Typ. Das heißt, sobald die sie betreffende Funktion aufgerufen ist, werden die Variablen im Speicher angelegt. Beim Beenden der Funktion löscht der Computer ihre Werte und gibt den benötigten Speicherplatz wieder frei. Dadurch gehen ihre Inhalte verloren und können beim nächsten Aufruf derselben Funktion also nicht weiterbenutzt werden.

automatic-Variablen werden durch die Angabe des Schlüsselwortes »auto« spezifiziert:

```
auto int a;
auto char b;
auto float xyz;
```

Wenn Sie keine anderweitige Angabe hinzusetzen, verarbeitet er die Variablen selbsttätig als »auto«. Statt der gezeigten Definitionen können Sie also auch schreiben:

```
int a;
char b;
float xyz;
```

Diese Notation sind Sie ja bisher schon gewohnt. Ein anderes Verhalten zeigen die static-Variablen. Einmal im Speicher angelegt, sind sie dann bis zum Programmende bei jedem Aufruf der Funktion, für den die Definition gilt, verfügbar. Ihr Wert bleibt also auch beim Beenden der Funktion erhalten. Zur Verdeutlichung hier ein Programmbeispiel, das das unterschiedliche Verhalten von auto- und static-Variablen zeigt:

```
main()
{
 sub(); /* 1.Unterprogrammauf-
         ruf */
 sub(); /* 2.Unterprogrammauf-
         ruf */
}
sub();
{
     auto int x=3;
     printf("Wert von x=%d ",x);
     x=x+1;
}
```

Die Bildschirmausgabe ist »3 3«. Bei jedem Aufruf wird x neu angelegt. Ersetzen Sie das Schlüsselwort auto durch static und schauen Sie, was passiert: »3 4«. Beim ersten Aufruf von sub() wird x der Wert 3 zugewiesen und auch auf dem Bildschirm ausgegeben. Danach erhöht der Befehl x=x+1 den Wert auf 4, und der Computer beendet die sub-Funktion. Beim zweiten Aufruf von x »weiß« der Computer noch, daß x den Wert 4 hat und gibt ihn auch aus. static-Variablen eignen sich also dazu, wichtige Daten bis zum nächsten Aufruf der Funktion zu »konservieren«.

Auf Grund dieser Eigenschaft müssen Sie in jedem Einzelfall entscheiden, ob Sie nun auto oder static vorziehen. Auto-Variablen haben den Vorteil, daß Speicherplatz nur für wirklich benötigte Variablen benutzt und dieser auch baldmöglichst wieder freigegeben wird. Die zugehörigen Mechanismen der Variablenverwaltung kosten allerdings sowohl Zeit als auch Programmspeicherplatz. statics erlauben damit kürzere und schnellere Programme.

Die Speicherklasse »register« weist den Compiler an, eine Variable direkt in einem der Register des Mikroprozessors abzulegen. Der Prozessor kann sie so durch einen einzigen Maschinenbefehl sehr schnell erreichen, was sich entsprechend positiv auf die Arbeitsgeschwindigkeit des Programms auswirkt. Die »herkömmlichen« Prozessoren wie Z80, 6502, 6809 und sogar der 68000 besitzen aber gar nicht genügend Register, um noch einige zusätzlich einem C-Programm zur Verfügung zu stellen. Die diversen C-Compiler für kleinere Computer geben deshalb bei diesem Befehl entweder eine Fehlermeldung aus (sehr selten), oder sie ignorieren die Anweisung und machen aus »register« »auto« (das ist bei Compiler-Autoren weitaus beliebter). Sie werden nun sicher fragen: Wenn ich jetzt schon Funktionen definieren kann, muß ich doch irgendwie zwischen den einzelnen Funktionen Daten austauschen können. Bloß wie?

Basic-Programmierer werden antworten: Kein Problem. Wir haben doch die, ach so praktischen, globalen Variablen. Nun definieren wir einfach alle Variablen als global und sind sämtlicher Sorgen über den Datenaustausch enthoben.

Vor diesen Gedankengängen muß aber dringendst gewarnt werden. Globale Variablen lassen sich vielleicht noch ohne Probleme bei kürzeren Programmen (nicht länger als drei oder vier Seiten) anwenden. Bei umfangreicherem C-Quellcode sind globale Variablen aber sehr gefährlich. Da eine Änderung ihres Inhalts auch Auswirkungen auf alle anderen Funktionen hat, sind die Folgen nicht mehr überschaubar.

Diese Nebeneffekte machen das ganze Konzept des modularen Aufbaus von C zunichte und führen über kurz oder lang ins (vor-)programmierte Chaos.

C bietet viel zuverlässigere Wege zur Parameterübergabe und -übernahme. Parameter haben Sie bereits in Ihrem ersten Programm übergeben. Erinnern Sie sich an »printf(»Hallo, Welt!")«? Die Stringkonstante »Hallo, Welt!« ist nichts anderes als ein Parameter, der an die Funktion printf abgeschickt wird. C-Funktionen können eine beliebige Anzahl von Parametern beliebiger Datentypen (int, char, float und so weiter) übernehmen, zum Beispiel:

```
putchar(64);
printf("TEXT");
putc(3,6);
func(3,a,'*',b,1);
```

Die Funktionen kopieren sich die Argumente in sogenannte »formale Parameter«. Das sind nichts anderes als lokale Variablen, in die der Inhalt der angegebenen Parameter übertragen wird. Die formalen Parameter sind also nicht die Variablen selbst, sondern sie repräsentieren nur ihren Wert. Eine Veränderung ihres Wertes variiert folglich nicht die Variablen, die als Argumente in der aufrufenden Funktion benutzt werden.

Schreiben wir ein Programm, das Integer-Daten aus dem Hauptprogramm in eine Unterfunktion übernimmt und auf dem Bildschirm ausgibt:

```
main()
{
 int i; /* Parameter-
        Variable vereinbaren */
 i=4711;
 drucke(i);
 drucke(4711);
}
 drucke(x) /* x ist formaler
            Parameter*/
 int x; /* Parametertyp
         definieren */
{
 printf("%d\n",x);
}
```

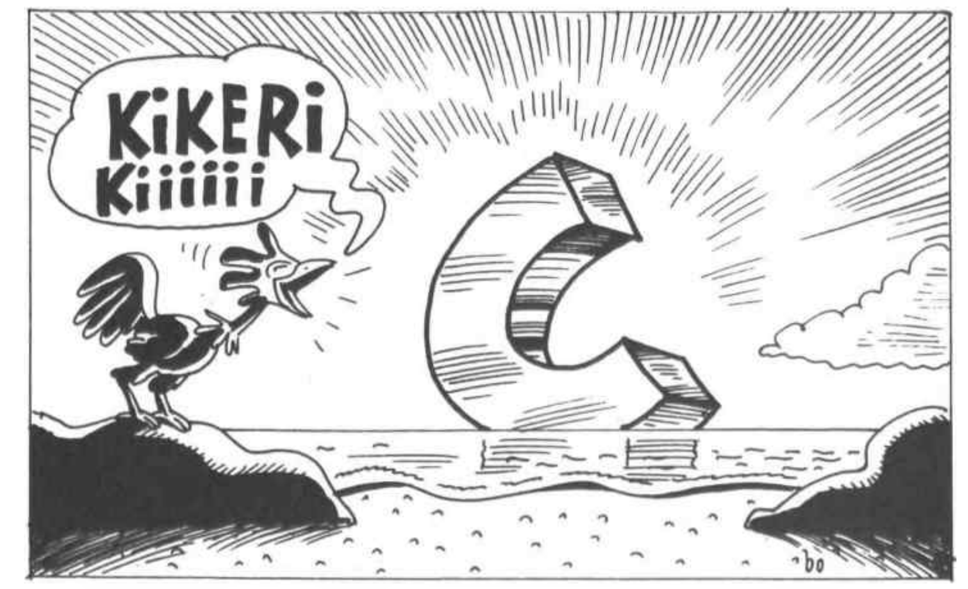
Der Aufruf »drucke(i);« führt dazu, daß der Computer sich den Wert von i holt

(nicht i selbst!) und an das lokale x in drucke(x) übergibt. »drucke(4711)« weist den Computer an, die Konstante 4711 dem lokalen x zuzuweisen.

Ebenso lassen sich zwei, drei oder eine beliebige andere Zahl von Parametern übergeben:

```
main()
{
 int x,y,z;
 x=y=23;
 y++;
 z=-x;
 drucke3var(x,y,z);
}
 drucke3var(a,b,c)
 int a,b,c;
{
 printf("%d %d %d",a,b,c);
}
```

Nebenbei ist an diesem Programm die Zeile »x=y=23« recht interessant.



Dieser Befehl weist den Variablen x und y gleichzeitig den Wert 23 zu. Das läßt sich mit beliebig vielen Variablen machen: »a=b=c=d=e=f=44«. Diese Form der Wertzuweisung darf nicht mit den Logiktests in Basic verwechselt werden, wo A=B=3 die Variable B mit 3 vergleicht und abhängig davon A auf TRUE (-1) oder FALSE (0) setzt.

Ist Ihnen aufgefallen, daß die Variablendefinition in drucke3var(a,b,c) nicht, wie bisher üblich, in die geschweiften Klammern des Programmcodes integriert ist? Der Grund liegt darin, daß die zu den formalen Parametern korrespondierenden Definitionen außerhalb der Klammern stehen müssen. Die übrigen »normalen« Variablendefinitionen sind wie gehabt in die Klammern der Verbundanweisung zu setzen:

```
main()
{
 druckschleife('*',70);
 druckschleife(33,70);
 druckschleife('-',20);
}
druckschleife(zeichen,wiederhol)
char zeichen;     /* formale
                     Parameter */
int  wiederhol;
{
int zaehler;    /* normale
                   lokale
                   Variable */
for (zaehler=0; zaehler<
     wiederhol; zaehler++)
     putchar(zeichen);
putchar('\n');
}
```

Der Funktion »druckschleife(zeichen,wiederhol)« fällt die Aufgabe zu, das Zeichen »zeichen« »wiederhol«-mal zu drucken. druckschleife('*',70) gibt also 70 Sternchen auf dem Bildschirm aus. Die Variablen zeichen und wiederhol sind formale Parameter und werden deshalb außerhalb der geschweiften Klammern definiert. zaehler ist eine interne lokale Variable; die Definition erfolgt daher innerhalb der Blockklammern.

Die Datenübergabe geht nun problemlos vor sich. Anders sieht es mit der Übernahme von Werten aus der aufgerufenen in die übergeordnete Funktion aus.

Prinzipiell gibt es in C zwei Arten von Funktionsaufrufen:
- funktion(parameter);
- variable=funktion(parameter);

Im ersten Fall werden nur Daten an die Funktion übermittelt, ein Rückgabewert wird nicht erwartet. Wohin auch damit? Damit entspricht dieser Fall gewissermaßen den Prozeduren in Pascal.

Im Gegensatz dazu erwartet man beim zweiten Funktionsaufruf, daß die Unterfunktion irgendeinen Wert in die Variable einträgt. Dabei kann das return-Statement Verwendung finden.

Ein Hinweis an die Basic-Programmierer unter Ihnen: Eine der beiden Formen des return-Befehls ist identisch mit dem Basic-Kommando RETURN (Rücksprung aus Unterprogrammen).

Das normale return-Statement beendet die Ausführung einer Funktion und bewirkt die Rückkehr ins aufrufende Programm. Sie können aber auf die Anwendung verzichten, da die Rückkehr sowieso von der geschlossenen geschweiften Klammer ausgelöst wird:

```
main()
{
 funktion();
}
 funktion();
{
 printf("Hallo, hier bin ich!");
 return; /* kann entfallen! */
}
```

Als recht nützlich erweist sich aber return, um zum Beispiel, abhängig von einer if-else- oder switch-case-Konstruktion, die Bearbeitung einer Funktion zu beenden:

```
main()
{
 int i;
 i=getchar(); /* Tastaturzeichen
                 lesen */
 check_small(i); /* Pruefen, ob
          zwischen 'a' und 'z' */
}
 check_small(x)
 int x;
{
 if (x>='a' && x<='z') { printf
      ("Liegt zwischen a und z");
                          return;
}
 printf("Kein Kleinbuchstabe!");
}
```

Das Programm liest ein Zeichen von der Tastatur und prüft mit der Funktion check_small, ob es sich um einen Kleinbuchstaben handelt. In diesem Fall gibt der Computer »Liegt zwischen a und z« aus und bricht die Bearbeitung der Funktion check_small ab. Andernfalls meldet er »Kein Kleinbuchstabe!« und beendet ebenfalls die Funktion.

Die erweiterte Form return(x) schickt den Inhalt der Variablen x an die Variable, die beim Funktionsaufruf zu finden ist. Eine umständliche Formulierung für die Zuweisung »i=5« sieht in C so aus:

```
main()
{
 int i;
 i=zuweisfunkt();
 printf("%d",i);
}
 zuweisfunkt()
{
 return(5); /* 5 an die auf-
            rufende Funktion */
}
```

»int i« weist den Computer an, i als Integervariable zu verstehen. »i=zuweisfunkt()« bewirkt den Funktionsaufruf, und printf sorgt für die Bild-

schirmausgabe, anhand derer Sie die Arbeitsweise von return überprüfen können. »zuweisfunkt()« enthält als einzigen Befehl die return-Anweisung, die den konstanten Wert 5 zurückgibt.

Meistens handelt es sich wohl vor allem um die Zusammenarbeit zwischen der Übernahme und der Rückgabe von Parametern. Die normale C-Zuweisung »variable=ausdruck;« kann auf diese Weise mit einem kleinen Programm ersetzt werden, durch »variable=calc(ausdruck);«. Im Programmieralltag ist die gezeigte Routine natürlich völlig überflüssig – es sei denn, Sie wollen auf Biegen oder Brechen den Speicher füllen:

```
main()
{
 int i,j;
 i=calc(3);       /* i=3 */
 j=calc(i*2);     /* j=i*2 */
 i=calc(i+i*j/2); /* i=i+i*j/2 */
 printf("i=%d j=%d",i,j);
}
 calc(x)
 int x; /* formaler Parameter */
{
 return(x);
}
```

Es gibt noch zwei Sonderfälle, die bei der Programmierung auftreten können:

– Im Programmtext steht ein einfaches return ohne Variable oder return fehlt völlig. Der an die Variable (links vom Gleichheitszeichen) zugewiesene Wert ist dann ungültig und ergibt keinen Sinn.

– Ein Wert wird mittels return(x) zurückgegeben. Da aber die Funktion keine Zuweisungsvariable vorfindet (zum Beispiel printf("x") statt y=printf("x")), geht der Wert verloren. Er richtet dann aber nicht irgendwelche undefinierbaren Schäden an, wie man durchaus erwarten könnte, wenn ein Wert in einen nicht dafür vorgesehenen (und daher eventuell verbotenen Bereich) geschrieben wird.

Bisher sind wir stillschweigend stets davon ausgegangen, daß alle Funktionen Integer-Argumente liefern. Dem ist aber nicht unbedingt so. Sie können auch Zeichen, Fließkommawerte oder Leerargumente übergeben. Dazu werden die Funktionen ähnlich wie die Variablen deklariert.

```
funktion(x)
 char x;
{
 putchar(x);
}
```

können Sie gleichwertig schreiben:

```
int funktion(x)
 char x;
{
 putchar(x);
}
```

Immer wenn Sie keine derartige Angabe machen, nimmt der Compiler an, daß Sie eine Integer-Funktion deklarieren wollen. Eine Funktion, die einen float-(Fließkomma-)Wert an das aufrufende Programm übergibt, sieht so oder ähnlich aus:

```
float pi_wert()
{
 return(3.141592653);
}
```

Mit printf("%f",pi_wert()) läßt sich der Wert für Pi von einem anderen Programmteil aus aufrufen.

Neben den üblichen Definitionen gibt es noch »void«. Schnell im Englisch-Wörterbuch geblättert – dort stehen unter anderem als Übersetzung »leer« und »frei«. void zeigt dem Compiler an, daß die Funktion keinen Wert übermittelt, also eine »leere« Wertrückgabe hat:

```
void func(x)
 int x;
{
 putchar(x);
}
main()
{
 func(64);
}
```

Die Angabe von »void« ist aber eher für penible Programmierer gedacht und auch nicht in allen Compilern implementiert. Denselben Effekt erzielen Sie, wenn Sie kein »void« angeben und die Funktion statt dessen so programmieren, daß sie von selbst auf die Rückgabe von Werten verzichtet.

Wir haben vorhin festgestellt, daß die Position von Funktionen relativ zur Lage der main-Funktion völlig beliebig ist. Eine Einschränkung muß aber für die Fälle gemacht werden, die Funktionen ausdrücklich mit einem Typenmerkmal deklarieren.

Entdeckt der Compiler während des Übersetzens von main() den Aufruf einer von Ihnen definierten Funktion, kann er nicht erkennen, welchen Typ diese Funktion hat, und nimmt deshalb automatisch das am häufigsten gebrauchte »int« an. Ist der Funktionswert tatsächlich Integer, gibt es keine weiteren Probleme. Bei den anderen Typen wie »char«, »float« und »double« meldet der Compiler einen Fehler, weil er diese nicht mit seiner vorherigen Annahme in Einklang bringen kann.

Dieses Problem läßt sich auf zwei Wegen lösen:

– Sie stellen alle Funktionen, die nicht Integer sind, im Programmlisting irgendwo vor den Ort, an dem sie das erste Mal aufgerufen werden. Der Compiler hat dann intern einen Vermerk gespeichert, welchen Typ die Funktion hat:

```
char zeicheneingabe()
{
 return(getchar());
}
main()
{
 char tastenzeichen;
 printf("Bitte geben Sie ein
       Zeichen ein: ");
 tastenzeichen=zeicheneingabe();
 printf("Das Zeichen war ein %c",
       tastenzeichen);
}
```

Die Funktion »zeicheneingabe()« gleicht der getchar()-Funktion völlig. Sie gibt daher auch nur den Wert zurück, den sie von getchar() erhalten hat. In der Hauptfunktion main() druckt der Computer eine Aufforderung aus, ein Zeichen einzugeben.

»tastenzeichen=zeichengabe()« holt sich das Zeichen, und printf gibt es auf dem Bildschirm aus. Hätten Sie die Funktion zeicheneingabe() hinter main() gestellt, würde sich Ihr C-Compiler mit einer hämischen Fehlermeldung »bedanken«.

– Die zweite Möglichkeit ist die sogenannte »Vorwärtsdeklaration«. Diese läßt sich in eine globale und eine lokale Vorwärtsdeklaration trennen. Zuerst zur globalen: Sie setzen einfach an den Anfang Ihres Programms eine Definition des Funktionsnamens, die einer Variablendefinition ähnelt. Der Compiler erkennt daran den Typ der Funktion und kann sie richtig übersetzen:

```
char zeicheneingabe(); /* Vor-
           wärtsdeklaration */
main()
{
 char tastenzeichen;
 printf("Bitte geben Sie ein
       Zeichen ein: ");
 tastenzeichen=zeicheneingabe();
 printf("Das Zeichen war ein %c",
       tastenzeichen);
}
 char zeicheneingabe()/* Tatsäch-
     liche Funktionsdefinition */
{
 return(getchar());
}
```

In der Vorwärtsdeklaration dürfen keinerlei formale Parameter wie »char zeicheneingabe(x)« oder »char zeicheneingabe(x,'t',3)« angegeben werden. Sie endet daher immer mit leeren runden Klammern. Erst bei der eigentlichen Definition der Funktion sind die Parameter wie gehabt einzutragen:

```
char zeichenausgabe();
          /* Ohne Parameter! */
main()
{
 char x;
 x=zeichenausgabe(65);
 printf("\nASCII-Code %d",x);
}
 zeichenausgabe(z)
 /* Hier Parameter z angeben! */
 char z;
 /* Und Parameter deklarieren */
{
```

```
 putchar(z);
 return(z);
}
```

Die Vorwärtsdeklaration in der ersten Programmzeile enthält keinerlei Parameter. Erst bei der Funktionsdefinition zeichenausgabe(z) ist der Parameter angegeben. Die Funktion hat die Aufgabe, ein Zeichen auf dem Bildschirm auszugeben und dessen ASCII-Code als char-Variable an die aufrufende Funktion zurückzusenden. Dort kann er dann ausgewertet werden.

Was Sie schon bei der Besprechung der globalen Variablen gehört haben, wiederholt sich auch hier: Die globale Vorwärtsdeklaration ist nicht so empfehlenswert, wie sie auf den ersten Blick vielleicht aussieht. Besser ist es, in jeder Funktion, die die zu definierende Funktion aufruft, die Vorwärtsdeklaration einzusetzen. Das hat den Vorteil, daß Sie immer den Überblick haben, welche Funktion andere Funktionen benutzt:

```
main()
{
 char zeichenausgabe();
        /* Lokale Deklaration */
 char x;
 x=zeichenausgabe(65);
 printf("\nASCII-Code %d",x);
}
zeichenausgabe(z)
 /* Hier Parameter z angeben! */
 char z;
 /* Und Parameter deklarieren */
{
 putchar(z);
 return(z);
}
```

Ein C-Compiler bietet Ihnen in Form eines Vorübersetzers, des »Pre-Processors«, zusätzliche Dienste an. Der #include-Befehl fügt andere C-Quellcode-Dateien in den Programmtext ein.

Der Präprozessor (so die deutsche Bezeichnung) ist eine Art Textaustausch- und Einsetzprogramm. Er hat »absolut keine Ahnung« von der Sprache C und dient nur dazu, die von Ihnen eingegebenen Quellcode-Dateien in eine vom Compiler lesbare Form zu bringen. Alle Präprozessor-Kommandos beginnen mit dem Doppelkreuz-Zeichen »#«. Sie sind nicht genormt, sondern vom Compiler abhängig. Wichtig für die alltägliche Programmierarbeit ist neben #include noch #define. Die übrigen Befehle wie #if, #ifdef und #ifndef dienen der bedingten Compilierung, die die meisten von Ihnen wahrscheinlich nie brauchen werden.

Die Anweisung #define gestattet es, Texten oder Zahlen im Programm sinnvolle Namen zu geben. Eine Zeile wie

```
umfang=2*radius*3.141592653
```

enthält die Zahl Pi als Konstante. Wenn Sie diesen Wert häufiger in Ihrem Programm brauchen, werden Sie die folgende Vereinfachung schätzen. Der Präprozessor erlaubt Ihnen, folgendes zu schreiben:

```
#define PI 3.141592653
umfang=2*radius*PI;
flaeche=radius*radius*PI;
```

Statt 3.141592653 darf also nach dem #define-Befehl synonym der Ausdruck »PI« benutzt werden. Beim Durchlaufen des Präprozessors ersetzt dieser automatisch alle »PI«s durch den Zahlenwert.

Der Präprozessor entfernt also stur die alten Bezeichnungen und fügt ebenso stur die neuen Namen ein. Immerhin ist er so intelligent, dies nicht mitten in Zeichenketten zu machen:

```
#define TEXT ERSATZTEXT
printf("TEXT");
```

Dieses Programm gibt also trotzdem »TEXT« aus und nicht etwa »ERSATZTEXT«. In alter Gewohnheit schreiben die C-Programmierer solche #define-Namen in Großbuchstaben, damit sie sich besser von Funktionsaufrufen, Variablennamen und ähnlichem abheben. Wenn sich irgendwann einmal der Wert der Konstanten ändert (bei Pi ist das kaum anzunehmen, bei anderen Daten aber durchaus möglich), müssen Sie nur im Programmkopf (dort sollten Sie alle #define's der Übersichtlichkeit halber zusammenfassen) die Werte ändern.

Die Syntax für den #define-Befehl sieht so aus:

```
#define AUSDRUCK1 AUSDRUCK2
```

Die Zeile schließt also nicht wie bei üblichen C-Anweisungen ein Strichpunkt ab. Den AUSDRUCK1 und den AUSDRUCK2 muß mindestens ein Leerzeichen oder ein TAB-Code voneinander trennen. Im allgemeinen ist es anzuraten, den zweiten Ausdruck bei #define in Klammern zu setzen, wenn dieser eine arithmetische Formel darstellt. Stellen Sie sich die Probleme bei dieser Definition vor:

```
#define ZWOELF 3*4
i=48/ZWOELF;
```

Sie erwarten hier als Ergebnis sicher eine Vier. Doch weit gefehlt! Der Compiler macht daraus »i=48/3*4«. Und das ist 64, nicht 4.

Von Klammern umgeben, wird der Ausdruck richtig bearbeitet:

```
#define ZWOELF (3*4)
```

Das wird zu:

```
i=48/(3*4)
```

Hieraus berechnet der Computer:

```
i=48/12=4
```

Hier das gewünschte Ergebnis. Sie können sich bei #define auch auf bereits bestehende Ausdrücke beziehen. Das folgende Programm definiert die Variable b gleichzeitig als »a« und »b«. Sie können alle drei scheinbaren Variablennamen benutzen. Diese bewirken auch alle dasselbe:

```
#define a b
#define c a
main()
{
 int c;
 a=33;
 b=b+1;
 printf("%d",c);
}
```

Der Compiler empfängt vom Präprozessor den Text in folgender Form:

```
main()
{
 int b;
 b=33;
 b=b+1;
 printf("%d",b);
}
```

Bei vielen Compilern geht #define aber über einen reinen Textaustausch hinaus. Sie erlauben die Angabe von Argumenten, die mitübersetzt werden:

```
#define QUADRAT(x) (x*x)
 printf("%d; %d;%d",QUADRAT(3),
 QUADRAT(3.14),QUADRAT(0));
```

In die vom Compiler erwartete Form überträgt das der Präprozessor:

```
printf("%d %d %d",3*3,3.14*3.14,
      0*0);
```

Wenn der Präprozessor jetzt noch den Programmcode optimieren kann, ersetzt er die Berechnungen durch Konstanten:

```
printf("%d %d %d",9,9.8596,0);
```

Zwischen den Makronamen und das Argument dürfen Sie nie einen Leerraum oder ein TAB-Zeichen einfügen, denn in diesem Fall erkennt der Präprozessor das Argument als zweiten Ausdruck und bringt das ganze Programm durcheinander. Statt »#define QUADRAT (x) (x*x)« müssen Sie also »#define QUADRAT(x) (x*x)« schreiben.

»Wann Makros, wann Funktionen?« wird Ihre Frage lauten. Allgemein läßt sich sagen, daß Makros zu längeren, aber schnelleren Programmen führen. Schließlich werden sie an jeder Stelle im Programmtext eingesetzt. Funktionen sind zwar langsamer bei der Programmbearbeitung, sparen aber im allgemeinen Speicherplatz.

Der Präprozessor ist ein beliebter Ansatzpunkt für den Compiler-Hersteller, zusätzliche Routinen in den Übersetzer einzufügen, ohne gleichzeitig die Kompatibilität mit anderen Compilern zu verlieren. Beispiele für zusätzliche #-Kommandos sind etwa »#list-« beziehungsweise »#list+« zum Ein- und Abschalten der Programmauflistung während des Compilierens oder »#line« zum Hinzufügen von Zeilennummern in die Listdatei.

(Martin Kotulla/hg)

Die Dimensionen von C

Kaum ein Programm, das Daten jeglicher Art verwaltet, kommt ohne Datenfelder aus. Sie sind damit unverzichtbarer Teil jeder Programmiersprache – so auch von C.

```
#define LINEFEED 10              /* Systembedingter LF-Code */

main()
 {
  char zeichen[255];             /* Char-Array mit 255 Elementen */
  int  i;                        /* Variable für Zählschleifen */
  char c;                        /* Speicher für gelesene Zeichen */
  c=i=0;                         /* c und i initialisieren */

  while (c!=LINEFEED)            /* Lesen, bis c=Line-Feed */
   { c=getchar();                /* Ein Zeichen lesen */
      zeichen[i]=c;              /* In Zeichenarray übertragen */
      i=i+1;                     /* Indexzähler erhöhen */
   }

  i=i-1;                         /* Zähler korrigieren */
  while (i>=0)                    /* Ausgabeschleife */
   {
     putchar (zeichen[i]);     /* Zeichen aus dem Feld ausgeben */
     i=i-1;                      /* Zähler vermindern */
   }
 }
```

Listing 1. Aus »Regen« wird »negeR«

C kennt wie jede andere Programmiersprache Arrays, auf Deutsch Datenfelder. Sie werden ähnlich wie unter Basic angelegt. Diese sogenannte Dimensionierung gleicht der normalen Variablendefinition, mit dem Unterschied, daß die Dimension (Anzahl der Felder) in eckigen Klammern stehen muß. Ferner sind alle Felder explizit anzulegen.

```
int   a[23];
char  buchstabenfeld[255];
float kommazahlen[12];
```

Das erste Element trägt immer den Namen 0, das letzte einen Index (Subskript), der um eins niedriger ist als die angegebene Dimension. Ein Feld x[4] besitzt somit die Elemente x[0], x[1], x[2] und x[3]. Mit den Einzelvariablen eines Datenfeldes kann man genauso rechnen wie mit normalen Variablen. Wertzuweisungen wie »a[3]=233« sind ebenso erlaubt wie Formelausdrücke der Art »x[b]=x[b-1] *x[b+2]/3«. Felder des Typs char bieten die Simulation endlich auch Strings. Als Beispiel dafür steht das folgende Programm, das die Eingabe eines Satzes von der Tastatur erwartet und diesen dann in umgedrehter Reihenfolge wieder ausgibt. Es verwandelt beispielsweise den Satz »Lernen Sie doch C!« in die Wendung »!C hcod eiS nenreL«. Das Programm funktioniert aber bei Ihnen nur, wenn Ihr System mit einer gepufferten Ein- und Ausgabe arbeitet. Was das ist, haben wir ja schon weiter vorne dargestellt.

Das Programm (Listing 1) bestimmt mit #define eine Konstante LINEFEED. Diese repräsentiert den Wert 10. Da nicht jeder Computer auch den ASCII-Code 10 als das Zeichen versteht, das die Eingabe abschließt, braucht ein Programmierer nur diese Zeile anzupassen, um die Routine auf anderen Geräten zum Laufen zu bringen.

»char zeichen[255]« erklärt das Feldzeichen zu einem Array mit 255 Buchstaben. »int i« definiert die Variable i, die als Schleifenzähler benutzt wird. Die Variable, die dem Compiler mit char c bekanntgemacht wird, nimmt jeweils ein Zeichen von der Tastatur entgegen, bevor es in dem Feld gespeichert wird.

»c=i=0« setzt die Variablen c und i in einem Arbeitsgang auf den Wert Null. C-Variablen sind nämlich nicht wie in Basic unbedingt mit Null vorbesetzt; sie enthalten den Wert, der zufällig an ihrer Speicherstelle steht.

Die erste while-Schleife liest solange Zeichen von der Tastatur ein, bis sie einen Zeilenvorschub-Code entdeckt. In der Schleife erfolgt die Zeicheneingabe mit »c=getchar()«; »zeichen[i] =c« überträgt den ASCII-Code der gedrückten Taste in das Datenfeld. »i=i+1« sorgt dafür, daß der Indexzähler auf das Datenfeld erhöht wird.

Nach der Schleife muß das Programm mit »i=i-1« den Zeigerwert korrigieren, da dieser immer am Schleifenende inkrementiert wird, was jetzt wieder rückgängig gemacht werden muß.

Die zweite while-Schleife gibt die einzelnen Zeichen des Arrays mit putchar in umgekehrter Reihenfolge aus, bis i den Wert 0 annimmt. In diesem Augenblick ist das Programm beendet.

C verzichtet bei den Arrays auf eine Überprüfung des Wertebereichs der Indexzähler. Sie können also ohne Fehlermeldung von einem Feld mit 10 Elementen das zwanzigste Element auslesen und auch einschreiben:

```
main()
 {
   int a[10];
   printf("%d",a[20]);
 }
```

Das C-Programm gibt dann einfach den Wert aus, der an der Adresse steht, die es intern aus der Indexnummer errechnete. Das wirft natürlich die Frage auf, wie C intern die Arrays abspeichert. Doch dazu erfahren Sie erst später mehr, wenn wir ausführlich auf den Umgang mit Zeigern (»pointer«) eingehen.

Denn es gibt noch einiges mehr zu sagen über die Datenfelder selbst. So sind Sie zum Beispiel nicht auf eindimensionale Arrays beschränkt. Sie können auch mit zwei- oder dreidimensionalen Feldern arbeiten. Die maximale Dimension wurde von Kernighan und Ritchie nicht exakt vorgeschrieben. Die meisten Compiler erklären sich auch mit fünf Dimensionen noch einverstanden.

Die zusätzlichen Dimensionen werden durch Anhängen weiterer Indizes in eckigen Klammern kenntlich gemacht:

```
main()
  {
    int a[2][3];
    int i,j;
    printf("i j a\n\n");
    for (i=0; i<=1; i=i+1)
      {
        for (j=0; j<=2; j=j+1)
          {
          a[i][j]=i+j;
          printf("%d %d %d\n",
          i,j,a[i][j]);
          }
      }
  }
```

Das Programm schreibt in ein (2,3)-dimensionales Integer-Datenfeld mit Hilfe zweier ineinander geschachtelter Laufschleifen jeweils die Summe der Werte von i und j. Anschließend gibt es diese deutlich sichtbar auf dem Bildschirm aus.

Genauso können Sie das mit mehr als zwei Dimensionen machen:

```
char  a[4][5][1];
float flt[10][20][30][40]);
```

In C dürfen Sie die Feldgröße, die in der Variablendefinition angegeben ist, nicht mit einer Variablen bezeichnen. Dort muß immer eine Konstante stehen, denn C erlaubt keine dynamische Verwaltung von Datenfeldern. Das Programm

```
main()
  {
    int a=5;
    int b[a];
  }
```

funktioniert also unter C nicht. Es führt immer zu einer Fehlermeldung und zum Abbruch der Compilierung.

Wir lernten vorher einen Weg kennen, Variablen bei der Definition mit einem Anfangswert zu initialisieren. Das ist auch bei den Datenfeldern möglich – mit der Einschränkung, daß auto-Felder da nicht mitmachen. Gestattet ist das logischerweise nur bei den statics. Automatic-Variablen werden ja bei jedem Funktionsaufruf neu angelegt. Da sie nach Bearbeitung der Funktion jedesmal gelöscht werden, woher soll dann der Computer beim erneuten Aufruf die alten Werte nehmen? Die Lösung ist relativ einfach.

Eine Vorbesetzung von eindimensionalen Arrays mit Werten programmieren Sie so:

```
main()
{
  static int feld[8]=
  {2,4,8,16,32,64,128,256};
  int i;
  for (i=0; i<8; i=i+1)
    printf("%d %d\n",i,feld[i]);
}
```

Das Programm setzt in die Elemente des Arrayfeldes der Reihe nach die Zahlen 2, 4, 8, 16, 32, 64, 128 und 256 ein. Die nachfolgende Ausleseschleife beweist, daß das funktioniert. Sie gibt den Schleifenzähler und das Feldelement mit dem betreffenden Index aus.

Wenn Sie keine Lust haben, nachzuzählen wieviele Werte Sie angegeben haben, um daraus den Wert in den eckigen Klammern zu erhalten, können Sie diese stumpfsinnige Arbeit auch dem Compiler überlassen. Er macht das gerne für Sie:

```
main()
  {
    static int feld[]=
    {2,4,8,16,32,64,128,256};
    int i;
    for (i=0; i<8; i=i+1)
    printf("%d %d\n",i,feld[i]);
  }
```

Die Angabe der Standardwerte ist auch bei mehrdimensionalen Feldern erlaubt. Allerdings gestaltet sich die Schreibweise etwas komplizierter:

```
static int feld[3][4]={ {1,2,3,4},
            {4,3,2,1}, {5,4,3,2}};
```

In einem komplexen Programm schaut das so aus:

```
main()
  {
    static int x[3][4]={{1,2,3,4},
                        {4,3,2,1},
                        {5,4,3,2}
                           };
    int i,j; /* Schleifenzähler */
    for (i=0; i<3; i=i+1)
      { for (j=0; j<4; j=j+1)
          printf("%d %d %d\n",
          i,j,x[i][j]);
        putchar('\n');
      }
  }
```

```
main()
  {
    int feld[20];              /* Normale Definition des Arrays */
    int maxindex;              /* Variable für den maximalen Index */
    int i;                     /* Als Schleifenzähler */
    maxindex=20;               /* Höchste Indexnummer angeben */
    for (i=0; i<20; i=i+1) feld[i]=255-i;
    ausdrucken(feld,maxindex); /* Angabe ohne Klammern! */
  }
ausdrucken(array,index)       /* "feld" ohne Klammern! */
 int array[];                 /* Formaler Parameter, leere Klammern*/
 int index;                   /* Formaler Parameter für Feldgröße */
 {
   int i;                     /* Lokaler Schleifenzähler */
   for (i=0; i<index; i=i+1) printf("%d\n",array[i]);
 }
```

Listing 2. Datenfelder verschiedener Größe

Es handelt sich wieder um zwei ineinander verschachtelte Schleifen, die alle Feldelemente zusammen mit den Werten von i und j ausdrucken. Nach jeweils vier Zeilen, also wenn die innere Schleife einmal abgearbeitet ist, gibt putchar (\) einen Wagenrücklauf und einen Zeilenvorschub aus, um das Ergebnis übersichtlicher zu gestalten.

Jetzt benötigen wir nur die Information, wie sich Datenfelder als Argumente an Funktionen übergeben lassen.

Da wir vorher feststellten, daß die Größe von Arrays konstant zu bleiben hat, müssen wir das bei der Übergabe an Funktionen wieder einschränken. Sollte man für jede Feldgröße etwa eine eigene Funktion schreiben? Natürlich nicht. Sie übergeben der Funktion den Namen des Feldes ohne nachfolgende eckige Klammern, und in den meisten Fällen sinnvollerweise eine Information über die Größe des Arrays. Im Funktionskopf ist der Feldname wieder ohne Klammern angegeben, bei der Definition der formalen Parameter mit leeren eckigen Klammern. Was sich jetzt furchtbar umständlich anhört, sieht in einem C-Programm sehr übersichtlich aus (siehe Listing 2).

Datenfelder groß und klein

Das Integerarray feld[20] enthält 20 Elemente, die Variable maxindex wird mit dieser Zahl geladen. Die for-Schleife versieht alle Elemente mit Werten, die von der Funktion »ausdrucken(feld)« auf den Bildschirm gebracht werden.

Auf diese Art lassen sich Datenfelder unabhängig von ihrer Größe an Funktionen übergeben. Doch eines müssen Sie unbedingt beachten. Während bei normalen Variablen deren Wert übermittelt wird (»call by value«) und anhand dieses Wertes der Computer eine neue Variable anlegt, macht der Computer bei Funktionsaufrufen von Datenfeldern keine Kopie, sondern läßt die Funktion mit dem originalen Datenfeld arbeiten. Eine Manipulation von einem Wert der Feldelemente innerhalb der Funktion wirkt sich demnach auch auf das Datenfeld im aufrufenden Programm aus. Dies wird durch das Programm in Listing 3 belegt.

Obwohl also das Datenfeld in main() und in subfunc() unterschiedliche Namen hat (»feld« beziehungsweise »array«), ist es doch ein- und dasselbe Array. In der Hauptfunktion wird das zweite Element mit der Zahl 5 initialisiert, in subfunc() verändert der Computer den Wert auf 20. Nach der Rückkehr nach main() steht jetzt auch dort der Wert 20.

Wenn Sie sich schon einmal mit Maschinensprache auseinandergesetzt haben oder gar ein »Assembler-Profi« sind, wissen Sie sicher, was Zeiger (auf Englisch »pointers«) sind. Für Sie ist das folgende dann als Wiederholung gedacht. Die übrigen Leser erfahren einige interessante Neuheiten.

Stellen Sie sich eine ganz normale Integervariable vor. Mit printf können Sie ihren Wert ausgeben, mit Hilfe der verschiedenen Operatoren den Variablenwert verändern. Wenn Sie sich jetzt in die Lage des Computers versetzen, machen Sie sich klar, daß er sich ja irgendwo die Variableninhalte merken muß. Um sie auch wiederzufinden, legt er sie an einer Speicheradresse ab und verwaltet sie anhand dieser Adresse.

Ein Zeiger ist nun nichts anderes als eine Variable, die auf die Speicheradresse einer anderen Variablen hinweist. Hat Ihnen nun ein Zeiger die Adresse offenbart, können Sie den Variableninhalt indirekt beliebig manipulieren.

Sogar in verschiedenen Basic-Dialekten gibt es Funktionen, die die Adresse einer Variablen ausgeben. In MBasic und dem Basic-Interpreter des Atari ST ist das VARPTR(x), in Atari-6502-Basic ADDR(x) und im Schneider-Basic der Klammeraffe: PRINT $@a$.

Die Sprache C zeigt hier ihre Maschinennähe, denn das Zeigerkonzept wird vorzüglich unterstützt. Ganz ungefährlich ist das Arbeiten mit Zeigern aber nicht. Schnell gibt eine solche Variable eine falsche Adresse an, und das System verabschiedet sich mit allen intern gespeicherten Daten und Programmen auf Nimmerwiedersehen – beziehungsweise bis Sie die Reset-Taste drücken. Seien Sie also vorsichtig mit den Zeigern. Wenn Sie aber behutsam damit umgehen, vereinfachen sie die Programmierung oft erheblich.

Eine Sicherung bieten moderne C-Compiler gegen den allzu hemmungslosen Gebrauch von Zeigern. Die Variablen, die als Zeiger fungieren sollen, können den normalen Integervariablen nicht beliebig zugewiesen werden und bedürfen einer besonderen Art der Definition:

```
int   *integerzeiger;
char  *zeichenzeiger;
float *fliesskommazeiger;
```

Die Initialisierung ist also nur bis auf den vorangestellten Stern mit der normalen Variablendefinition identisch.

Der Stern spielt bei den Zeigeroperationen eine besondere Rolle, ebenso das kaufmännische Und-Zeichen »&«. Der Zeigertyp folgt dem Datentyp der Variablen, auf die er zeigt. Ein Zeiger auf einen Buchstaben hat also den Typ »char«, ein Zeiger auf eine Fließkommavariable den Typ »float« und so weiter.

```
main()
  {
    int array[3];
    array[2]=5;    /* In der Hauptfunktion mit 5 laden */
    printf("In main() hat array den Wert %d\n",array[2]);
    subfunc(array);
    printf("Jetzt auch in main(): array=%d",array[2]);
  }

 subfunc(feld)
  int feld[];
  {
    feld[2]=20;     /* In subfunc() mit 20 laden */
    printf("In subfunc() hat feld den Wert %d\n",feld[2]);
  }
```

Listing 3. Doppelte Datenmanipulation

Das &-Symbol liefert die Adresse einer Variablen in einer Zeigervariablen ab. Das funktioniert so:

```
main()
  {
    char zeichenvariable;
      /* normale Variable */
    char *zeichenzeiger;
      /* Pointervariable */
    zeichenvariable='#';
      /* Variable mit # laden */
    zeichenzeiger=
    &zeichenvariable;
      /* Pointer auf Variable */
    printf("%u",zeichenzeiger);
      /* Pointer ausdrucken */
  }
```

»zeichenvariable« wird als eine normale char-Variable definiert, »zeichenzeiger« als Zeiger auf eine char-Variable. Als nächstes weist das Programm der Variablen ein Symbol, das Doppelkreuz, zu. Die darauffolgende Zeile lädt einen Zeiger auf die Speicheradresse der Variablen in die Zeigervariable. Die printf-Zeile sorgt dafür, daß Sie die Adresse auch wirklich erfahren.

Die Formatoption »%u« in der printf-Funktion gibt einen arithmetischen Ausdruck in dezimaler Schreibung ohne Vorzeichen an (»unsigned«). Dies ist nötig, da manche Computer Adressen, die größer als +32767 sind, als negative Werte ansehen.

Doch was profitieren wir davon, daß wir jetzt die Adresse der Variablen kennen? Zugegeben: Noch nicht allzuviel, aber es kommt schließlich noch mehr. Der *-Operator ist die Umkehrung des &-Operators. Er liefert den Wert in der Adresse, auf die die Pointervariable zeigt.

Wenn also variable1 '#' ist und zeigergleich&variable1, dann erhalten Sie mit variable2gleich*zeiger das Zeichen »#«. Eine recht umständliche Wertzuweisung, die die Zeigermethode einbezieht, ließe sich dem C-Compiler so vorgeben:

```
main()
 {
  char zeichenvariable;
    /* normale Variable */
  char *zeichenzeiger;
    /* Pointervariable */
  char zweite_variable;
    /* zweite char-Variable */
  zeichenvariable='#';
    /* Variable mit # laden */
  zeichenzeiger=&zeichenvariable;
    /* Pointer auf Variable */
  zweite_variable=*zeichenzeiger;
    /* Adresseninhalt holen */
  putchar(zweite_variable);
    /* neue Variable ausdrucken */
 }
```

Die erste Zuweisung versieht »zeichenvariable« mit dem Zeichen »#«, die folgende Zeile legt einen Zeiger auf die Variable nach »zeichenzeiger« – wie gehabt. »zweite__variable = *zeichenzeiger« überträgt den Inhalt der Adresse, auf die der Zeiger zeigt, in die Variable. Das ist nichts anderes als das bekannte Doppelkreuz.

Brauchbar anwenden lassen sich die Zeiger zum Beispiel bei der Übernahme von Argumenten aus Funktionen. Sie erinnern sich sicherlich: Es wird nur der Wert von Variablen übergeben, anhand dem neue lokale Variablen angelegt werden. Eine Veränderung des Werts der lokalen Variablen hat keinerlei Auswirkungen auf die Variablen im aufrufenden Programm. Mit der Zeiger-Methode können Sie jetzt aber direkt die Variablen im Funktionsaufruf manipulieren. Sehen Sie das an einem Programm. Es nimmt ein Zeichen von der Tastatur an und wandelt es, wenn ein Großbuchstabe kommt, in einen kleinen um. Das soll mit Hilfe eines Funktionsaufrufes namens »lower« geschehen (siehe Listing 4).

Die Variable »zeichen« liest hier einen Buchstaben oder ein Symbol von der Tastatur ein. »zeiger« findet Verwendung als Speicher für die Variablen-



```
main()
  {
    char zeichen;              /* Zeichenvariable */
    char *zeiger;              /* Zeigervariable */
    zeichen=getchar();         /* Tastaturzeichen einlesen */
    zeiger=&zeichen;           /* Zeiger auf Buchstaben holen */
    lower(zeiger);             /* Funktionsaufruf über Zeiger */
    printf("%c\n",zeichen);    /* Geändertes Zeichen ausgeben */
  }

 lower(zeiger)                 /* Funktion "lower" */
  char *zeiger;                /* Pointer als formaler Parameter */
  {
   char zeichen;               /* Lokale Variable */
   zeichen=*zeiger;            /* Tatsächliches Zeichen holen */
   if (zeichen<'A') return;    /* Wenn kleiner 'A' Rücksprung */
   if (zeichen>'Z') return;    /* Wenn größer 'Z' Rücksprung */
   *zeiger=zeichen+32;         /* ASCII-Offset addieren */
  }
```

Listing 4. Aus groß mach klein

adresse. Anhand dieses Zeigers übergibt das Programm die Variable an die Funktion »lower«. Nach Beendigung der Funktion enthält »zeichen« den konvertierten Code. printf gibt den veränderten Variablenwert aus.

Die »lower«-Funktion arbeitet mit einem Zeiger als formalem Parameter. Sie definiert sich eine lokale Variable, die das Zeichen enthält. Der Computer stellt mit den beiden if-Entscheidungen fest, ob das Zeichen überhaupt ein Großbuchstabe ist. Wenn ja, addiert der Computer den Wert 32 zum ASCII-Code ('a'=97, 'A'=65, 65+32=97). Mit dem Zeigerbefehl »*zeiger=zeichen+32« bewirkt die Funktion auch die Änderung im Hauptprogramm.

Die Zeigervariablen eignen sich natürlich auch zum Rechnen. Besonders interessant sind die Inkrement- und Dekrementoperatoren. Sie zählen nicht mehr um 1 auf- oder abwärts, sondern um eine Einheit des Datentyps auf- oder abwärts. Zeiger auf char's verändern den Wert um ein Byte, Integerwerte um zwei Byte, long int's bei 32-Bit-Arithmetik vier Byte und Fließkommazahlen abhängig von der Genauigkeit zum Beispiel um sechs Byte.

Diese »Serviceleistung« von C wird am folgenden Programm deutlich:

```
main()
  {
    int  integer,*intzeiger;
    char character,*charzeiger;
    integer=16383;
    character='%';
    intzeiger=&integer;
    charzeiger=&character;
    printf("char: %u %u\n",
        charzeiger,++charzeiger);
    printf("int:  %u %u\n",
        intzeiger,++intzeiger);
  }
```

Wie das Programm zeigt, liegen die zwei Werte von »char« ein Byte, die von »int« jedoch zwei Byte auseinander.

Auch die Addition und Subtraktion von Integer-Werten ist bei Zeigern möglich. Die Werte werden hier ebenfalls auf die Datenbreite umgerechnet:

```
main()
  {
    int i,*izeiger;
    i=0;
    izeiger=&i;
    printf("1. %d\n",izeiger);
    izeiger=izeiger+3;
    printf("2. %d\n",izeiger);
  }
```

Die Differenz ist hier nicht drei Byte, sondern sechs. Dies kommt daher, daß Integers zwei Byte lang sind, und 2*3 ergibt nun einmal 6.

Zu den weiteren erlaubten Operationen mit Zeigervariablen gehören der Vergleich zweier Zeiger und die Subtraktion eines Zeigers von einem anderen. Dies ist aber nur möglich, wenn beide Zeiger demselben Datentyp angehören. Eine Subtraktion eines char- und eines float-Pointers funktioniert also nicht.

Wichtiger als bei einfachen Variablen sind die Zeiger bei Datenfeldern und Stringkonstanten. Arrays lassen sich so nicht nur durch die Angabe der Indizes bearbeiten, sondern auch durch einen direkten Zugriff:

```
main()
  {
    static char string[]="Dies
           ist ein Zeichenarray";
           char *zeiger;
    zeiger=&string[0];
    while (*zeiger != '\0')
           putchar(*zeiger++);
  }
```

Das Programm benutzt ein Zeichenarray mit dem Namen »string« und setzt einen Zeiger auf das erste Element. In einer while-Schleife gibt der Computer die Zeichen aus, bis er ein Byte mit dem Wert 0 entdeckt. Dies ist eine Besonderheit von Zeichenketten unter C. Sie werden der Reihe nach im Speicher als ASCII-Codes abgelegt. Das letzte Zeichen ist immer ein Nullbyte »\0«. Dies erleichtert die Bearbeitung von Strings über Zeiger erheblich mehr, als das Hantieren mit String-Deskriptoren in Basic.

Auf eine Kurzschreibweise sei auch noch hingewiesen. Statt »&string[0]« dürfen Sie auch einfach »string« schreiben:

```
main()
  {
    static char string[]=
           "Dies ist ein
           Zeichenarray";
           char *zeiger;
    zeiger=string;
    while (*zeiger != '\0')
           putchar(*zeiger++);
}
```

Bevor Sie jetzt darüber räsonieren, wieso man die Schreibweise ändern darf, wollen wir Ihnen sagen, daß dahinter keine Logik steckt, sondern schlicht die Tippfaulheit der C-Entwickler.

Mit Pointer-Variablen läßt sich auch ein Datentyp String simulieren, ohne daß direkt sichtbar wird, daß Sie hier in Wirklichkeit auf Datenfelder zurückgreifen:

```
main()
  {
    char *string;
    string="Dies ist ein
           String in C!";
    printf("%s",string);
  }
```

Eine neue Option für printf wird auch gleich mit vorgestellt: »%s« druckt einen String aus, auf den ein Zeiger deutet. Die Ausgabe wird von der printf-Funktion beendet, sobald sie ein Nullbyte, den Stringbegrenzer, findet.

Wenn Sie ein komplexes Programm entwickeln, benutzen Sie darin wahrscheinlich nicht nur die grundlegenden Datentypen wie Integerzahlen und einzelne Buchstaben, sondern auch kombinierte Datentypen. Ein Programm zur Verwaltung von Personendaten erwartet beispielsweise Datensätze mit folgendem Aufbau:

1. Identifikations-Nummer der Person
2. Geschlecht
3. Familienstand
4. Einkommen beziehungsweise Gehalt
5. Zahl der Kinder

Bisher fast ausschließlich eine Domäne von Pascal (Stichwort RECORDs), macht C mit »structures«. Das Schlüsselwort »struct« leitet die Definition eines kombinierten Datentyps ein. Unsere Personendatei ließe sich so darstellen:

```
struct
{
  int  idnummer;
  /* Identifikations-Nummer */
  char geschlecht;
  /* "w" beziehungsweise "m" */
  char familienstand;
  /* "v"=verheiratet, "l"=ledig*/
  int  einkommen;
  /* Einkommen bzw. Gehalt */
  int  kinderzahl;
  /* Zahl der Sprößlinge */
} person;
```

Sie sehen, Sie können in einer Struktur gleichzeitig verschiedene Datentypen verwenden. Hier sind das beispielsweise »int« und »char«.

Die Strukturdefinition setzt sich aus dem Schlüsselwort »struct«, der öffnenden geschweiften Klammer »{«, der Liste der Strukturkomponenten (hier »char geschlecht; char familienstand; ...«), einer schließenden geschweiften Klammer »}«, dem Namen der Struktur (hier »person«) und einem abschließenden Strichpunkt zusammen.

Im folgenden können Sie einzelne Teile der Struktur ganz normal wie Variable verwenden. Die Strukturkomponenten werden durch den Namen »strukturname.komponentenname« angesprochen. Erika Mustermann hätte dann vielleicht folgende Daten für ihren neuen Personalausweis:

```
person.idnummer=1984;
person.geschlecht='w';
person.familienstand='v';
person.einkommen=2800;
person.kinderzahl=2;
```

Mit den Strukturelementen lassen sich auch arithmetische und logische Operationen durchführen – etwa nach einer Gehaltserhöhung:

```
person.einkommen=person.einkommen
      +200;
printf("Einkommen von Person %d "
      ,person.idnummer);
printf("ist %d",person.
      einkommen);
```

C bietet noch eine Besonderheit: Sie können auf einfache Art eine »Strukturmaske« definieren und diese dann auf mehrere Strukturen anwenden – vergleichbar mit Formularvordrucken – die unverändert mehrmals benutzt werden. Dem C-Compiler machen Sie das so verständlich:

```
struct personenmaske
{
  int  idnummer;
  /* Identifikations-Nummer */
  char geschlecht;
  /* "w" beziehungsweise "m" */
  char familienstand;
  /* "v"=verheiratet, "l"=ledig*/
  int  einkommen;
  /* Einkommen bzw. Gehalt */
  int  kinderzahl;
  /* Zahl der Sprößlinge */
};
```

Dies hat den Vorteil, daß Sie verschiedene Strukturen mit dem gleichen Aufbau auf einmal definieren können:

```
struct politiker
  {
    int gehalt;
    int beruf;
  };

  struct politiker helmut_kohl;
  struct politiker johannes_rau;
```

In diesem speziellen Fall ist aber folgende Befehlsfolge vorzuziehen:

```
main()
  {
    struct
      {
        int gehalt;
        int beruf;
      } helmut,johannes;

      helmut.gehalt=11200;
      helmut.beruf=1;
      johannes.gehalt=7450;
      johannes.beruf=2;
      printf("Verdient Helmut
      %d?\n",helmut.gehalt);
      printf("Und Johannes etwa
      %d?\n",johannes.gehalt);
  }
```

Die angegebenen Gehälter sind natürlich frei erfunden.

Strukturen können selbst wiederum Datenfelder und weitere Strukturen enthalten. Doch die nötigen Programmierkenntnisse gehen weit über das Ziel dieses Einführungskurses hinaus.

Wir kommen langsam zum Schluß unserer Einführung. Sie haben inzwischen die Fähigkeit, recht anspruchsvolle Programme in C zu entwickeln. Doch die bloße Kenntnis der C-Befehle ist nicht alles, was man braucht, um C zu beherrschen. Von eminenter Wichtigkeit ist auch das Wissen um die C-Bibliothek. Darin sind alle für die Arbeit mit der Sprache notwendigen Funktionen abgelegt. Obwohl die Bibliotheken der einzelnen Compilerhersteller natürlich keinerlei Normung unterliegen, zeigen sich doch überraschende Ähnlichkeiten. Der Großteil der Compiler schließt sich den Bibliotheken der Unix-Systemumgebung von Kernighan und Ritchie an. Im folgenden werden Sie eine Reihe wichtiger C-Funktionen kennenlernen, die für erfolgreiches Programmieren einfach unerläßlich sind. Am Grad der Übereinstimmung mit Ihrer Compiler-Bibliothek erkennen Sie auch die Vollständigkeit Ihres Compilers.

Zuerst zu einigen Funktionen, die arithmetische Aufgaben durchführen und für verschiedene Typenkonversionen zur Verfügung stehen:

»x=abs(y)« liest in die Variable x den absoluten Betrag von y ein. Der Betrag einer Zahl ist ihr positiver Wert. Negative Zahlen werden also mit –1 multipliziert. Positive Werte bleiben unverändert. abs ist die C-Entsprechung der ABS-Funktion von Basic und Pascal:

```
printf("%d",abs(-3));          3
printf("%d",abs(3));           3
printf("%d",abs(0));           0
```

»x=sign(y)« holt sich das Vorzeichen von y und schreibt es nach x. Zahlen kleiner als 0 liefern x=–1, die Null selbst liefert eine Null, positive Zahlen melden den Wert 1:

```
printf("%d",sign(-3));        -1
printf("%d",sign(0));          0
printf("%d",sign(3));          1
```

»x=atoi(string)« verwandelt einen String in eine Integerzahl. »atoi« heißt ausgeschrieben »convert ASCII string to integer«. Der String kann entweder als char-Array definiert sein oder als Konstante im Funktionsaufruf angegeben werden. Die Funktion arbeitet wie ASC bei den meisten Basic-Interpretern. Sie überliest alle voranstehenden Leerzeichen, TAB-Aufrufe und Newline-Codes. Sobald atoi die erste Ziffer entdeckt, beginnt die Umwandlung. Diese wird durchgeführt, bis ein Zeichen erreicht ist, das keine Ziffer mehr darstellt. Vor der Zahl dürfen auch die Vorzeichen »+« und »–« stehen. Einige Beispiele für die Benutzung von atoi:

```
printf("%d",atoi("233"));      233
printf("%d",atoi("3Text"));      3
printf("%d",atoi("Text"));       0
printf("%d",atoi("-314X"));   -314
```

Eine ganze Reihe von Funktionen dient speziell der Manipulation von Strings, die aber alle eigentlich nur Arrays des char-Typs sind. »strcat (string1,string2)« verknüpft zwei Strings, (»cat« steht dabei für »concatenate«). Das Ergebnis der Verbindung wird im Bereich des ersten Strings abgelegt. Die Funktion prüft aber nicht, ob das Datenfeld wirklich groß genug ist für die neuen Daten. Im schlimmsten Fall überschreibt der erweiterte String andere Daten oder auch den Programmcode. Als Demonstration von strcat finden Sie ein kleines C-Programm, das zwei Strings definiert und dann zusammen ausdruckt. Beachten Sie dabei, das erste Feld durch die Angabe »34« groß genug zu machen, um auch den zweiten String sicher unterzubringen:

```
main()
  {
    static char string1[34]=
         "Dies ist eine ";
    static char string2[]=
         " Stringverkettung.";
    strcat(string1,string2);
    printf("%s",string1);
  }
```

strcpy (string1,string2) hingegen kopiert den zweiten String an den Anfang des ersten (»strcpy« heißt »string copy«). Dadurch, daß das Nullbyte, das alle Strings abschließt, auch



mit übertragen wird, erscheinen bei printf nicht die eventuell überzähligen Zeichen des ersetzten Strings. Im folgenden Programm zeigt der Computer aus diesem Grund nur das Wort »C-String«:

```
main()
  {
    static char string1[]=
      "Dies ist der alte String";
    static char string2[]=
      "C-String";
    strcpy(string1,string2);
    printf("%s",string1);
  }
```

Auch hier müssen Sie der strcpy-Funktion genügend Platz zur Erweiterung des Strings zugestehen.

strcmp(string1,string2) vergleicht zwei Strings und setzt abhängig vom Ergebnis die vorangestellte Variable (»strings compare«). strcmp ähnelt der Basic-Konstruktion »VERGLEICH=(A$=B$)« oder »IF A$=B$ THEN ...«. Die Funktion liefert einen negativen Wert, wenn der erste String kleiner ist als der zweite. Entsprechend ist der positive Wert definiert. Sind beide Strings identisch, ergibt die strcmp-Funktion den Wert 0:

```
printf("%d",strcmp("BC","A"));             1
printf("%d",strcmp("A","BC"));            -1
printf("%d",strcmp("gleich","gleich"));    0
```

strlen(string) ermittelt die Länge eines Strings und ist damit äquivalent zur LEN-Funktion in Basic. Leerstrings kennzeichnet der Wert 0.

```
printf("%d",strlen("C-Sprache"));          9
printf("%d",strlen(""));                   0
```

Drei Funktionen, die die Manipulation von Einzelbuchstaben gestatten, heißen »tolower« (Umwandlung in einen Kleinbuchstaben), »toupper« (Umwandlung in einen Großbuchstaben) und »toascii« (Umwandlung in ein ASCII-Zeichen durch Löschen des siebten Bits des Codes):

```
printf("%c",tolower("A"));                 a
printf("%c",toupper("b"));                 B
printf("%c",toascii(161));                 !
```

Verschiedene Funktionen testen Buchstaben und setzen daraufhin eine Variable auf 1 (TRUE) oder 0 (FALSE):

```
ISALNUM:
printf("%d",isalnum("F"));                 1
printf("%d",isalnum("3"));                 1
printf("%d",isalnum("#"));                 0
```

isalnum heißt »is it alpha-numeric?« und wird auf TRUE gesetzt, sofern das Symbol eine Ziffer oder ein Buchstabe ist.

```
ISALPHA:
printf("%d",isalpha("Z"));                 1
printf("%d",isalpha("3"));                 0
```

isalpha (»is it an alphabetic letter?«) ist TRUE, wenn das Symbol einen Buchstaben darstellt.

```
ISDIGIT:
printf("%d",isdigit("3"));                 1
printf("%d",isdigit("A"));                 0
```

isdigit (»is it a digit?«) meldet TRUE, wenn das Symbol eine Ziffer von 0 bis 9 ist.

```
ISASCII:
printf("%d",isascii(126));                 1
printf("%d",isascii(254));                 0
```

isascii (»is it an ASCII code?«) liefert den booleschen Wert TRUE, wenn der ASCII-Code des Zeichens kleiner als 128 ist. Zwischen 0 und 127 ist der ASCII-Code nämlich standardisiert, Werte von 128 bis 255 definiert jeder Computerhersteller nach Belieben:

```
ISGRAPH:
printf("%d",isgraph(200));                 1
printf("%d",isgraph(120));                 0
```

isgraph (»is it a graphic character?«) setzt TRUE, wenn der ASCII-Code des Zeichens größer als 127 ist. Dort liegen bei den meisten Computern Grafiksymbole.

```
ISCNTRL:
printf("%d",iscntrl(13));                  1
printf("%d",iscntrl(32));                  0
```

iscntrl (»is it a control character?«) übergibt TRUE, wenn das Zeichen einen Control-Code repräsentiert und demnach im Bereich ASCII 0 bis ASCII 31 liegt.

```
ISLOWER:
printf("%d",islower('a'));                 1
printf("%d",islower('A'));                 0
```

islower (»is it a lower character?«) testet, ob das Zeichen ein Kleinbuchstabe ist. Trifft dies zu, wird das Ergebnis auf TRUE gesetzt.

```
ISUPPER:
printf("%d",isupper('Z'));                 1
printf("%d",isupper('z'));                 0
```

isupper (»is it an upper character?«) prüft, ob das Zeichen ein Großbuchstabe ist. In diesem Fall ist das Ergebnis TRUE.

```
ISPRINT:
printf("%d",isprint('*'));                 1
printf("%d",isprint(3));                   0
```

isprint (»is it a printable character?«) meldet TRUE, wenn das Zeichen auszudrucken ist. Dazu muß sein ASCII-Code zwischen 32 und 126 liegen (127 ist der Delete-Code).

```
ISPUNCT:
printf("%d",ispunct(','));                 1
printf("%d",ispunct('7'));                 0
```

ispunct (»is it a punctuation character?«) stellt fest, ob es sich um ein druckbares Zeichen handelt, das nicht gleichzeitig eine Ziffer oder ein Buchstabe ist. Das wären zum Beispiel das Komma, das Ausrufezeichen, das Dollarsymbol und der »Klammeraffe«.

```
ISSPACE:
printf("%d",isspace('\t'));                1
printf("%d",isspace('\n'));                1
printf("%d",isspace(32));                  1
printf("%d",'?'));                         0
```

isspace (»is it a space character?«) prüft nach, ob es sich bei dem char-Symbol um ein Leerzeichen, TAB-Code oder Newline (\) handelt. Diese Zeichen ergeben TRUE, alle anderen FALSE.

Damit wären die grundlegendsten C-Funktionen besprochen. Wenn Sie die eine oder andere nicht in Ihrer Bibliothek finden können, ärgern Sie sich nicht allzu sehr darüber. Die meisten lassen sich nämlich sehr einfach simulieren. Zum Beispiel die abs-Funktion durch

```
abs(n)
    int n;
    {
      if (n<0) return(n);
               else return(n);
    }
```

Sicher entdecken Sie aber in der Library Ihres Compilers eine Vielzahl weiterer nützlicher Funktionen, für deren Erklärung hier einfach der Platz fehlt. So haben wir beispielsweise alle Funktionen zur Dateibehandlung ausgespart. Allerdings differieren diese auch bei den verschiedenen Computern. Denken Sie allein schon an die unterschiedlichen Vorschriften zur Bildung von Dateinamen.

Wenn Ihr Interesse für C geweckt ist, und Sie auch die letzten Feinheiten der Sprache kennenlernen wollen, sollten Sie sich ein entsprechendes Fachbuch kaufen. Mit der wachsenden Begeisterung für C in Amerika und Europa schwillt auch die Bücherflut an.

Auf ein Buch sei aber hier schon hingewiesen. »Programmieren in C« enthält Informationen direkt von den C-Schöpfern Kernighan und Ritchie, sozusagen direkt von der Quelle. Bei einem Disput, was in C erlaubt ist und was nicht, kann man dieses Standardwerk bedenkenlos als Referenz heranziehen. Für Anfänger (die Sie nach diesem Kurs aber eigentlich nicht mehr sind), ist das Buch aber mit Sicherheit zu schwere Kost. Es ist eben von Systemprogrammierern für Systemprogrammierer geschrieben. Wenn Sie aber C beherrschen, werden Sie auf die glasklar dargestellten Informationen sicherlich dankbar zurückgreifen. Seltsamerweise fehlen im gesamten Buch die Bindestriche in zusammengesetzten Substantiven in den Texten – Satzfehler oder eigenartiges Sprachverständnis der Übersetzer? Jedenfalls sind manche Sätze recht schwer zu verstehen. Ansonsten ist die Übersetzung aber gut gelungen.

(Martin Kotulla/hg)

Info: Brian W. Kernighan und Dennis M. Ritchie, »Programmieren in C«, Hanser-Verlag München/Wien, 1983, ISBN 3-446-13878-1. Wer die amerikanische Originalausgabe vorzieht: »The C Programming Language«, Prentice-Hall, 1977.

Gut sortiert ist halb gewonnen

Will man verschiedene Datentypen sortieren, sind auch unterschiedliche Sortierroutinen nötig. Unser Sort-Programm in der Sprache C kann mehrere Datentypen verkraften und somit eine Datei flexibler und universeller gestalten.

Das Sortieren von Daten ist eine Aufgabe, die in einer Vielzahl von Programmen vorkommt. Es gibt mehrere Sortieralgorithmen. Da aber meistens große Datenbestände sortiert werden müssen, ist es sehr wichtig, den für die jeweilige Aufgabe optimalen Sort zu verwenden.

Leider herrscht bei vielen Programmierern der Drang, Bibliotheksfunktionen an die jeweilige Aufgabe angepaßt zu schreiben. Besser und im Endeffekt zeitsparender ist es, universell einsetzbare Funktionen zu programmieren. Hat man, wie in unserem Fall, eine universelle Sortierroutine geschrieben, so kann sie in jedes Programm problemlos eingebunden werden, ohne daß erneuter Programmieraufwand anfällt.

Zu beachten ist besonders, daß die Sortierroutine nicht nur einen bestimmten Datentyp verwenden kann, sondern flexibel gehalten ist.

Ein Beispiel hierzu: Eine Lieferantendatei soll nach der Postleitzahl sortiert werden. Das ist sehr schnell programmiert. Möchte man aber die Liste eine Woche später nach den Namen ordnen, fällt für die Sortierroutine neue Programmierzeit an. Dies kann dadurch vermieden werden, daß die Routine für beliebige Datentypen einsetzbar ist. Sie kann dann in die eigene Bibliothek aufgenommen und in neue Programme eingesetzt werden. (Heiner Etzler/hi)

```
/* Diese Sortierfunktionen enthalten nur den eigentlichen
   Sortieralgorithmus.
   Fuer eine komplette Sortierfunktion muss der Programmierer
   zwei Funktionen selbst schreiben: comp() und swap().
   comp() vollzieht den Vergleich der Datenelemente, swap()
   ist fuer das Vertauschen zustaendig.
   Beide Funktionen muessen den behandelten Datentyp unbedingt
   beruecksichtigen.
   Das Sortieren von Daten ist eine Aufgabe, die in den meisten
   Programmen vorkommt. Es lohnt sich Sortiermodule fuer den
   eigenen Bedarf zu schreiben. */


/*********************************************************************/
/*           **********  Funktion Bubble-Sort  **********           */
/*********************************************************************/


/* Die Verwendung von Zeigern erlaubt, einen datensatzunabhaengigen
   Sortieralgorithmus f}r n Datensaetze zu programmieren. */

#define void int


void bubsort(n,comp,swap)
/* An diesem Haupteingangspunkt der Funktion Bubble-Sort werden
   drei Parameter uebergeben. */

unsigned n;
/* Der erste Parameter (unsigned) ist die Anzahl zu sortierender
   Datensaetze */

int (*comp)();
/* Der zweite Parameter ist ein Zeiger auf die Funktion, die alle
   Schluessel zweier Datensaetze vergleicht und einen Ganzzahlwert
    uebergibt. */
```

```
int (*swap)();
/* Der dritte Parameter zeigt auf die Funktion, die zwei Datensaetze
   vertauscht.

{

     unsigned t;
     /* t bestimmt, ob eine Vertauschung erfolgte. */

     unsigned j;




     /* j wird in Index-Records verwendet. */

     do {
     /* In dieser Funktion wird festgestellt, ob mindestens zwei
        Records existieren. Es wird mindestens ein Vergleich gemacht. */

           t = 0;
           /* Das swap-flag(Vertauschung) erhaelt den
              Anfangswert FALSE (0). */


           for(j = 0; j < n - 1; ++j)
           /* Datensatznummern von 0 bis n-1 */

                       if((*comp)(j, j + 1) > 0) {
                       /* compare (Vergleichsfunktion) wird mit den
                          Zaehlern der beiden Datensaetze als Parameter
                          aufgerufen. */

                          (*swap)(j,j + 1);
                          /* Hier wird das Vertauschen ausgefuehrt. */

                          t = 1;
                          /* Nach einer Vertauschung wird das
                             swap-flag auf TRUE (1) gesetzt. */
                       }
     } while(t);
       /* Der Sortiervorgang wird solange ausgefuehrt, bis keine
          Vertauschung mehr notwendig ist. */

}

/*        ********** Ende Funktion Bubble-Sort ********** */


/* Der Bubble-Sort ist einer der einfachsten Sortiermethoden. Er ist
   jedoch nicht sehr effizient, da bei jedem Schleifendurchlauf jeder
   Datensatz der zu sortierenden Records mit allen anderen verglichen
   werden muss. Bei n zu sortierenden Datensaetzen muessen n hoch 2
   Vergleichsoperationen durchgefuehrt werden - bei gro~em
   Datenbestand eine sehr zeitaufwendige Methode.
   Eine Alternative zum Bubble-Sort ist der sogenannte
                         Shell-Sort.
   Hier wird eine aus n Elementen bestehende Liste in kleinere Teil-
   listen unterteilt, die dann miteinander verglichen werden. */
```

Listing. Verschiedene Sortieralgorithmen in C realisiert

Fortsetzung Seite 100

```
/***************************************************************************/
/*                 **********  Shell-Sort  **********                    */
/***************************************************************************/


#define void int


void shsort(n,comp,swap)
/* Haupteinstiegspunkt Shell-Sort mit der Uebergabe von drei
   Parametern. */

unsigned n;
/* Anzahl der zu sortierenden Datensaetze */

int (*comp)();
/* Zeiger auf die Funktion, die zwei Datensatzschluessel
   vergleicht. */

int (*swap)();
/* Zeiger auf die Funktion, die zwei Records vertauscht. */
{
int m;
/* Intervallanzeiger */

int h, i, j, k;
/* Zaehler (intern) */


m = n;
/* Anfangswert f}r Intervall wird gesetzt */


while(m /= 2) {
/* Ist m /= 2 gleich Null, so ist der Sortiervorgang
   beendet. */

      k = n - m;
      /* k wird initialisiert. */

      j = 1;
      /* j wird initialisiert. */


      do {
      /* In dieser Schleife wird der untere Segment-
         grenzwert gesetzt. */

           i = j;
           /* i wird initialisiert. */


           do {
           /* Mit dieser Schleife wird der untere
              Segmentgrenzwert gesetzt. */

                h = i + m;
                /* Mittleren Grenzwert setzen. */
```

```
            if((*comp)(i - 1, h - 1) > 0 {
            /* Zeiger auf die Vergleichsfunktion setzen. */

               (*swap)(i - 1, h - 1);
               /* swap wird nach Vergleich aufgerufen. */

               i -= m;
               /* Wurde vertauscht, wird hier i gesetzt. */

        } else

               break;
               /* Ausstieg aus der Schleife */

} while(i >= 1);
  /* Uberpruefen von i auf weiteren Durchlauf */

j += 1;
/* j wird f}r weitere Vergleiche inkrementiert. */

} while(j <= k);


                    /* Ende aeussere Schleife */




            }

}

/* **********  Ende Funktion Shell-Sort  *********





/* Bubble-Sort und Shell-Sort sind fuer fast alle Sortierprobleme
   die Loesung.
   Ein weiterer bekannter Sortieralgorithmus ist der sogenannte
   Quick-Sort. Dieser Sort hat eine noch hoehere Leistungsfaehigkeit
   als der Shell-Sort.
   Das folgende Listing des Quick-Sort ist ein sehr schoenes Beispiel
   f}r rekursive Programmierung.

*/


/**********************************************************************/
/*                **********  Quick-Sort  **********                  */
/**********************************************************************/



#define void int



static int(*_comp)(), (*_swap)();
```

Listing. Verschiedene Sortieralgorithmen in C realisiert (Fortsetzung)

```
/* Zwei statische Variablen werden dazu benutzt, die Funktionszeiger
   in einer Reihenfolge zu speichern, die es erlaubt, die Zahl
   der an die rekursive Funktion uebergebenen Parameter zu
   erniedrigen. */


void qusort(n,comp,swap)
/* Haupteinstiegspunkt f}r Quick-Sort mit der Uebergabe von drei
   Parametern. */


unsigned n;
/* n enthaelt die Anzahl der zu sortierenden Datensaetze. */


int (*comp)();
/* Zeiger auf die Vergleichsfunktion */

int (*swap)();
/* Zeiger auf die Vertauschfunktion */

{

     _comp = comp;
     /* Die statische Variable comp wird auf den Parameter
        comp gesetzt. */

     _swap = swap;

     /* Die statische Variable swap wird auf den Parameter
        swap gesetzt. */

     _quick(0, n - 1);
     /* Die rekursive statische Funktion quick wird mit einem
        unteren Grenzwert von 0 und einem oberen Grenzwert von n-1
        ausgefuehrt. */

}

/* Ende der Funktion */


static void _quick(lb, ub)

unsigned lb, ub;

{

      unsigned j;

      unsigned _rearr();


      if(lb < ub) {

              if(j = _rearr(lb, ub))

                     _quick(lb, j - 1);

              _quick(j + 1, ub);
```

Listing. Verschiedene Sortieralgorithmen in C realisiert (Schluß)

```
        }
}

/* Ende der Funktion */


static unsigned _rearr(lb, ub)
unsigned lb, ub;
{

        do {
                while(ub > lb && (*_comp)(ub, lb) >= 0)
                        ub--;
                if(ub != lb) {
                    (*_swap)(ub, lb);
                    while(lb < ub && (*_comp)(lb, ub) <= 0
                            lb++;
                    if(lb != ub)
                        (*_swap)(lb, ub);
                }
        } while(lb != ub);
        return lb;
}

/* Ende der Funktion */

/*            **********  Ende Quick-Sort  *********


/* Als Indizes wurden vorzeichenlose Variable (unsigned) zur
   Definition der Unter- und Obergrenze des abzusuchenden Bereiches
   verwendet. Variable dieses Typs erweitern den Zahlenbereich,
   der innerhalb der Funktion angewendet werden darf.

   Manche Prozessoren begreifen Arithmetik nur ohne Vorzeichen.
   Werden hier nun normale Integers (int) benutzt, wird bei der Code-
   Erzeugung automatisch ein Vorzeichen gesetzt.
   In C kann der weit groessere Teil von Schleifenkontrollvariablen
   vorzeichenlos definiert werden.
```

Rückkehr einer alten Dame: Eliza

»Eliza« gehört zu den Klassikern unter den Computerspielen: Der Computer schlüpft in die Rolle eines Psychoanalytikers. Damit es nicht in Vergessenheit gerät, finden Sie hier eine Version in der hochaktuellen Sprache C.

Die Originalversion von Eliza programmierte Joseph Weizenbaum am MIT (Massachusetts Institute of Technology in Boston, USA) urspüglich in Lisp. Schon bald folgten Übersetzungen in Basic und Veröffentlichungen in verschiedenen Computerzeitschriften. Obwohl im Laufe der Zeit einige Vereinfachungen vorgenommen wurden, ist das Resultat doch noch interessant. Der Vorteil dieser Version ist, daß die Texte und Schlüsselwörter, mit deren Hilfe Eliza antwortet, nicht Bestandteil des Programms, sondern in einer eigenen Datei erfaßt sind. So können Sie ohne Neucompilierung Änderungen vornehmen und deren Auswirkungen studieren. Denkbar wäre so sogar eine Eliza-Version, die Deutsch spricht: Wenn Sie das Programm nur durch die Angabe des Namens »Eliza« starten, arbeitet es automatisch mit den Texten in der Datei Eliza.DAT. Wenn Sie aber beim Aufruf noch einen Dateinamen »Eliza <Texte>. DAT« angeben, werden die Texte aus dieser Datei genommen. Das Programm beenden Sie durch die Eingabe von »bye«.

Das Programm sucht zunächst in der Eingabe des Benutzers nach Schlüsselwörtern, die es kennt. Die Schlüsselwörter gleicher Bedeutung sind dabei in Gruppen zusammengefaßt, und zu jeder Gruppe gibt es eine Gruppe mit zum Schlüsselwort passenden Standard-Antwortsätzen. Wenn Eliza ein Schlüsselwort aus einer Gruppe findet, nimmt sie aus der korrespondierenden Gruppe einen beliebigen Standardantwortsatz. Je häufiger ein solches Wort typischerweise auftritt, desto länger sollte die Antwortliste sein – so wird oftmaliges Auftreten immer derselben Antwort vermieden.

Sobald ein Standardantwortsatz gefunden wurde, erfolgt eine weitere Umformung. Enthält er ein Sternchen »*«, so wird an dieser Stelle ein Nebensatz eingefügt. In der Standard-Eliza.DAT treten Sternchen nur am Ende eines Satzes auf. Sie dürfen sie aber auch in der Satzmitte verwenden.

Der Nebensatz entsteht aus der Eingabe des Benutzers. Der Eingabeteil, der dem Schlüsselwort folgt, ist Ausgangspunkt. Bevor er in die Antwort eingesetzt wird, wird er aber noch konjugiert: also Wörter wie »my« durch »your« ersetzt und umgekehrt.

Der so entstandene modifizierte Antwortsatz erscheint dann endgültig auf dem Bildschirm.

Leerzeilen in der Datei werden bei der Auswertung übergangen. Ein Schrägstrich »/« gilt bei der Auswertung ganz genau wie ein echtes Zeilenende als »logisches Zeilenende«. Die Datei Eliza.DAT ist in mehrere Gruppen unterteilt, die jeweils Zeilen mit dem Inhalt »$« voneinander trennen.

Die erste Gruppe enthält das Titelbild, das der Computer beim Start ausgibt. Die zweite Gruppe – die nur aus einer einzigen Zeile bestehen darf – präsentiert die Begrüßung, mit der das Programm erstmals eine Eingabe vom Benutzer anfordert.

Die dritte Gruppe beinhaltet paarweise Schlüsselwörter für die Konjugation eines Nebensatzes, die sich gegenseitig ersetzen. »i« muß in »you« und »you« in »i« umgesetzt werden. Deshalb bilden »i« und »you« ein Paar. Ab der vierten Gruppe bis zum Dateiende stehen paarweise jeweils eine Schlüsselwortgruppe und die dazu passende Antwortgruppe.

Beachten Sie, daß die Einhaltung dieser Regeln nicht überprüft wird. Wenn die Datei nicht richtig aufgebaut ist, »verhaspelt« sich Eliza und gibt nur noch Unsinn aus. Im schlimmsten (allerdings seltenen) Fall übersieht die Sortierfunktion sogar die Dateiendmarkierung, so daß das Programm abstürzt. Alle Schlüsselwörter und Konjugationen müssen klein geschrieben werden – sonst erkennt sie das Programm niemals.

Probleme bei der Antwort

Die Ergebnisse, die Eliza liefert, hängen stark von der Reihenfolge des Dateiaufbaus ab. Schlüsselwörter am Anfang der Tabelle haben Vorrang, unabhängig von der Stellung der gefundenen Wörter im Text. In der Regel ist es sinnvoll, Schlüsselwörter, die zu konjugierten Antworten führen, zuerst zu nennen, weil das die interessanteren Antworten ergibt. Sie können aber auch mit anderen Reihenfolgen experimentieren.

Es muß sichergestellt sein, daß in jeder Eingabe ein passendes Schlüsselwort gefunden wird. Jede Eingabe (bei Eliza auch in einer leeren!) enthält nur ein Leerzeichen. Das letzte Schlüsselwort der Tabelle ist deshalb ein einzelnes Leerzeichen (nicht verwechseln mit einer Leerzeile!).

Im Schlüsselwort »think« der Standarddatei Eliza.DAT ist die Buchstabenfolge »hi« enthalten. »hi« fungiert aber gleichzeitig als eigenes Schlüsselwort mit höherer Priorität – als Abkürzung von »hello«. Ohne weitere Vorkehrungen wäre so das Schlüsselwort »think« unauffindbar. Bestandteil des Schlüsselwortes »hi« ist deshalb ein Leerzeichen, um einen Fehlgriff auszuschließen. Ähnliche Probleme treten auch bei anderen Wörtern auf. Leerzeichen in der Datei haben also immer eine wichtige Bedeutung – Sie sollten ganz besonders darauf achten. Wenn ein Schlüsselwort nicht in der ersten Spalte beginnt, ist das ein Indiz dafür, daß ein wichtiges Leerzeichen voransteht. Leerzeichen am Ende einer Zeile werden durch Anhängen eines Schrägstriches »/« am Ende der Zeile sichtbar gemacht. Da der Schrägstrich ein logisches Zeilenende bedeutet, Leerzeichen aber beim Einlesen nicht beachtet werden, schadet das bei der Interpretation der Datei durch Eliza nichts.

Bei umfangreicheren Dateikonstrukten können bestimmte Präpositionen in mehreren Paaren der Konjugationstabelle auftreten. Wenn eine mehrfache Präposition bei der Analyse eines Paares in den Nebensatz eingefügt wurde, würde sie bei der Analyse des anderen Paares fälschlicherweise wieder entfernt und durch eine dritte ersetzt. Konjugations-Schlüsselwörter, die in dieser Weise eingesetzt werden, dürfen Sie deshalb in der Datei mit einem zusätzlichen Unterstreichungszeichen »_« versehen. Das verhindert ein erneutes Finden. Vor dem Einfügen des Nebensatzes in den gesamten Antwortsatz werden alle Unterstreichungszeichen automatisch entfernt.

Scheuen Sie sich nicht, die Datei Eliza.DAT nach eigenem Gutdünken zu

verändern (natürlich sollten Sie eine Sicherheitskopie aufbewahren!). Interessant ist, außer einer allgemeinen Erweiterung, etwa die Übersetzung der Texte ins Deutsche, mit Anpassung der Konjugationen oder ein ganz anderes Gesprächsthema. Vielleicht können Sie die Antworten auch so geschickt wählen, daß der Benutzer beeinflußt wird, über ein Thema zu sprechen, zu dem Eliza besonders viele Schlüsselwörter weiß?

Die vorliegende Version wurde unter CP/M-80 mit einem BDS C-Compiler compiliert und getestet. Mit geringen Änderungen läuft Eliza aber auch unter anderen Betriebssystemen und mit anderen Compilern. Da möglichst einfache Standardfunktionen verwendet werden, ist diese Eliza-Version auch mit den kleinsten Compilern zu verarbeiten.

Eliza paßt sich an

Anpassen müssen Sie die Aufrufe der Funktionen »fopen« und »getc« (zum Einlesen der Datei). Beim BDS C-Compiler erwartet die Funktion »fopen« beim Aufruf die Adresse des Dateinamens und die Adresse eines genügend großen Pufferspeichers. Die Funktion »getc« erwartet nur die Adresse des Pufferspeichers der gewünschten Datei. Die meisten Compiler verarbeiten aber andere Strukturen. Wenn Sie noch unerfahren sind, sollten Sie sich dazu ein einfaches Programm aus Ihrer »C-Bibliothek« ansehen, und die darin enthaltenen Dateioperationen im Eliza.C-Programm nachbauen. In der Regel lauten die neuen Versionen dann so:

»...if((fd = fopen(name, "r")) == NULL) {...«

zum Dateieröffnen und

»...getc(fd)...« oder ... »fgetc(f1)...«

zum Lesen eines Zeichens aus der Datei. Natürlich müssen Sie dann deklarieren »int *fd« und können die Deklarierung von »puffer[BUFFSIZ]« weglassen.

Sie ersparen sich graue Haare, wenn Sie Experimente zur Diskettenverwaltung immer auf einer leeren Diskette durchführen (mit Sicherheitskopie des bearbeiteten Programms auf einer anderen Diskette).

»CPMEOF« ist der Wert, der das Erreichen des Dateiendes signalisiert und heißt bei manchen Compilern einfach »EOF«, die Konstante »ERROR« manchmal kurz »ERR«. Wenn andere verwendete Konstanten in Ihrem »stdio.h« nicht definiert sind, können Sie sie auf einen beliebigen, Ihnen sinnvoll erscheinenden Wert festlegen.

Die Funktion »getns« unterscheidet sich von der Standard-Bibliotheksfunktion »gets« nur dadurch, daß beim Einlesen von der Tastatur die maximal erlaubte Zeilenlänge vorgegeben werden kann. Sie funktioniert allerdings nur unter CP/M-80. Wenn Sie keinen CP/M-Rechner haben, dürfen Sie den Aufruf »getns(eingabe, 77)« ohne Schaden einfach durch »gets(eingabe)« ersetzen. Dann müssen Sie eben auf den Komfort der automatischen Begrenzung der Zeilenlänge verzichten. Die Standard-Bibliotheksfunktion »bdos()« heißt manchmal auch »__bdos()« oder »Ubdos()« und ruft – ohne Einflußnahme des C-Systems – Betriebssystemfunktionen direkt auf.

Wenn der Compiler keine »Zeiger auf Zeiger« erlaubt, können die Variablendeklarationen »char **name« durch »int *name« ersetzt werden. Die Bedeutung und Struktur der Variable ändert sich dadurch nicht. In der Funktion selbst können doppelte Sternchen trotzdem weiterverwendet werden. Die Anwendungen dieser Variablen brauchen deshalb nicht geändert werden. Dasselbe gilt für Zeigerarrays; »char *name[]« kann ohne Schaden ebenfalls durch »int *name« ersetzt werden.

Manche Compiler hängen an jeden puts-Befehl automatisch einen Zeilenvorschub an. Sie merken das sofort an den vielen Leerzeilen auf dem Bildschirm. In diesem Fall streichen Sie im Quelltext einfach die Zeilen mit »\n«.

Die Konstanten DATEILLAENGE, MAXZEILEN und MAXKEYS bestimmen, wieviele Bytes im Speicher für die Datei Eliza.DAT reserviert werden. Wenn Sie nur wenig Speicher haben, wählen Sie etwas kleinere Zahlen.

DATEILAENGE bestimmt in etwa die maximal erlaubte Dateilänge, hier also 20000 Byte. MAXZEILEN gibt an, wieviele Zeichen die Datei maximal haben darf. Beachten Sie jedoch, daß jede Zeile zwei Nummern benötigt. Wenn Sie also in der Datei 1000 Zeilen erwarten, müssen Sie MAXZEILEN auf den Wert 2000 definieren. MAXKEYS gibt die maximale Zahl Schlüsselwortgruppen – Antwortgruppenpaare an. Jede Schlüsselwortgruppe belegt allerdings drei Nummern! Mit einer Definition von »#define MAXKEYS 400« sind also nur etwa 130 Schlüsselwortgruppen erlaubt.

Das Innenleben einer Dame

Zum Vergleich: Die hier gedruckte Grunddatei Eliza.DAT benötigt mindestens die Werte DATEILAENGE = 4500, MAXZEILEN = 520, MAXKEYS = 100. Ohne Änderung der Daten können Sie also bis zu viermal längere Wortschatzdateien verwenden. Eine Überschreibung der zulässigen Maximallängen prüft das Programm allerdings nicht. Diese führt ohne Warnung zum Systemabsturz.

Der Hauptaufwand des Programms liegt darin, leistungsfähige Routinen zur Zeichenkettenverarbeitung zur Verfü-

```
/*
    beim Aufruf mit 'ELIZA' wird automatisch 'ELIZA.DAT' geladen. Durch
    Aufruf mit 'ELIZA <dateiname>' kann stattdessen eine andere Datei
    verwendet werden. Der korrekte Dateiaufbau und erlaubte Dateilaenge
    wird aber nicht ueberprueft !
    Die Funktion 'getns' muss an das Betriebssystem angepasst werden !
    Der Aufruf von 'fopen' in funktion 'einlesen' variiert von Compiler
    zu Compiler ===> unbedingt anpassen !!!
*/

#include "stdio.h"

#define DATEILAENGE 20000  /* maximale Dateigroesse in Bytes */
#define MAXZEILEN    2000  /* 2 mal die maximale Zeilenzahl der Datei */
#define MAXKEYS       400  /* 3 mal die maximalen Schluesselwortgruppen */

/*
    'getns' unterscheidet sich von der Standardbibliotheksfunktion 'gets  nur
    durch die Angabe der maximalen Zeilenlaenge und laueft nur unter CP/M-80.
    In anderen Systemen Funktionsrumpf einfach durch 'gets(eingabe)'ersetzen !
*/

getns(eingabe, maxlen)
char eingabe[];
int maxlen;
{
    char puffer[MAXLINE + 2];

    *puffer = maxlen;                   /* maximale Laenge in erstes Pufferbyte  */
    bdos(10, puffer);                   /* Aufruf CP/M - Zeileneditor            */
    puffer[2 + puffer[1]] = NULL;       /* tatsaechliche Laenge im 2. Pufferbyte */
    strcpy(eingabe, puffer + 2);        /* Text ab zweitem Pufferbyte            */
}

/*
    tlen ist  tatsaechliche Laenge des Strings text, slen ist Laenge von such.
    'instr' sucht in 'text' ab Position 'start' nach dem String 'such'. Wenn
    gefunden Rueckgabe der Position von 'such', sonst '0'.
*/
instr(start, text, tlen, such, slen)
```

Listing 1. Smalltalk mit Eliza

gung zu stellen. Der Aufwand lohnt sich aber. Die Routinen »getns«, »instr«, »einlesen« und »holeeing« sind sehr universell und auch für andere Programme brauchbar. Am besten fügen Sie sie in Ihre Standard-Bibliotheksfunktionen ein (kopieren Sie aber vorher Ihre Standard-Bibliothek). Da die Funktionen keine globalen Variablen benötigen, ist das ohne weiteres möglich.

Die Funktion »getns(eingabe,maxlen)« entspricht der Funktion »gets(eingabe)« mit zusätzlicher Angabe der Zeilenlänge, bei deren Überschreibung die Eingabe abgebrochen wird. Intern stützt sich die Funktion auf den CP/M-Zeileneditor. Deshalb ist sie leider nur unter CP/M-80 einsetzbar.

Die Funktion »einlesen(name,datei, descript)« liest die Datei, deren Name ab der Adresse »name« gespeichert ist, von der Diskette und legt sie im Speicher ab der Anfangsadresse »datei« (am besten ein Charakterarray) ab. Ein einziges Nullbyte ersetzt mehrfache Kombinationen von CR (Wagenrücklauf) und LF (Linefeed). Dadurch wird jede Zeile in einer eigenen Zeichenkette abgelegt, und betriebssystem- und compilerabhängige Unterschiede zwischen Textdateien werden beseitigt. Bei manchen Systemen genügt nur LF oder CR, um eine neue Zeile zu signalisieren, andere Systeme (zum Beispiel CP/M und MS-DOS) benötigen beide. Als Nebeneffekt werden bei dieser Manipulation alle Leerzeichen entfernt.

Im Programm Eliza gilt auch ein Schrägstrich »/« als Zeilenvorschubbyte. Bevor Sie diese Funktion in die Standardbibliothek aufnehmen, sollten Sie eventuell diese Abfrage entfernen. Wenn Sie mit mehreren Dateien zugleich arbeiten wollen, schließen Sie die Datei nach dem Einlesen, um den benötigten Diskettenpuffer wieder freizugeben. Sonst wird mit der Zeit das System verstopft.

Als Drittes wird der Funktion ein Integerfeld übergeben. In dieses werden während des Einlesens – analog zur Stringverarbeitung in Basic – Stringdescriptoren abgelegt. Jeweils zwei aufeinanderfolgende Feldelemente enthalten Informationen zum selben String. Das erste Feldelement eines Paares enthält die Anfangsadresse einer Zeile und das zweite dessen Länge. Um die Länge einer Zeichenkette festzustellen, braucht jetzt also nicht mehr die ganze Zeichenkette nach einem Nullbyte durchsucht zu werden. Das erste nicht mehr benötigte Element des Descriptorfeldes wird mit Null markiert. Zur Verwaltungsvereinfachung kann im Hauptprogramm das Descriptorfeld auch als Array definiert werden, dessen Elemente analog zu

```
char text[], such[];
int start, tlen, slen;
{
    int i, ende;
    char *txtp, *txt2p, *suchp, *such2p, c;

    if(slen == 1) {          /* Sonderfall 1-Zeichenstring (Geschwindigkeit! */
        c = *such;
        for(txtp = text + start - 1; *txtp != NULL; txtp++)
            if(*txtp == c)
                return txtp - text + 1;
        return 0;
    } else {
        ende = tlen - slen + 1;  /* letzte Position, an der such Platz hat */
        txtp = text + start - 1; /* Anfang nicht beachten */
        suchp = such + 1;
        for(i = start; i <= ende; i++) {  /* moegliche Anfangspos. von suc */
            if(*such == *txtp++)  /* Kontrollschleife nicht immer n|tig */
                for(such2p = suchp, txt2p = txtp; *such2p == *txt2p; txt2p++)
                    if(*++such2p == NULL)  /* wenn Ende von such, dann o.k */
                        return i;
        }
        return 0;
}

/*
    Datei von Diskette einlesen; dabei mehrfache Zeilenuebergaenge (CR, LF und
    Leerzeilen) ersetzen durch ein einziges 'NULL'. Zeichen '/' gelten als
    logische Zeilenuebergaenge. Zusaetzlich Aufbau eines
    Stringdescriptorenfeldes: je 2 aufeinaderfolgende descript-Elemente
    enthalten Adresse und Laenge einer Zeile
*/

einlesen(name, datei, descript)
char name[], datei[];
int descript[];
{
    char *dateip, puffer[BUFSIZ];     /* 'puffer' ist Datenpuffer zum Diskio */
    int c, *zeilp, len;               /* '*zeilp'  Zeiger in Descriptorenfeld*/

    if(fopen(name, puffer) == ERROR) {        /* Hier Anpassung an verwendeten*/
        puts("***** Datei '");                /* Compiler notwendig !         */
        puts(name);
        puts("' nicht gefunden *****\n");
        exit();
    }
    zeilp = descript;
    *zeilp++ = dateip = datei;                /* Adresse der ersten Zeile    */
    len = 0;                                  /* Startwert fuer Zeilenlaenge */
    while((c = getc(puffer)) != CPMEOF) {     /* Lesen, bis EOF auftritt     */
        if(c == ERROR) {
            if(*(dateip - 1) != NULL) {       /* Unerwartetes Dateiende wie  */
                *dateip++ = NULL;             /* Zeilenende behandeln        */
                *zeilp++ = len;
                zeilp++;
            }
            break;
        }
        if(c == 13 || c == 10 || c == '/') {  /*  Newlinemarkier. der Datei*/
            if(*(dateip - 1) != NULL) {        /* Leerzeilen nicht beachten */
                *dateip++ = NULL;
                *zeilp++ = len;                /* Laenge in Stingdescriptor  */
                *zeilp++ = dateip;             /* Adresse naechste Zeile     */
                len = 0;
            }
         } else {
            *dateip++ = c;                     /* 'normales' Zeichen         */
            len++;
         }
    }
    *dateip = EOF;                             /* Ende der Datei             */
    *--zeilp = 0;                              /* Auch im Descriptorfeld     */
}

/*
    Datei fuer Verwendung durch Eliza vorbereiten: Anfangsadressen der
    Begruessungszeile, der Konjugationsregeln, Schluesselwortgruppen bestimmen
    Nach jeder Schluesselwortgruppe folgt ein Satz dazupassender Antworten
*/

ordnen(descript, servus, konjfirst, keylist)
int descript[], *servus, *konjfirst, keylist[];
{
    char **zeilp;
    int *keyp, i;

    zeilp = descript;           /* Header - zeilen ueberspringen                */
    while(**zeilp != '$')       /* '$' ist Zeichen fuer Gruppenende             */
        zeilp += 2;
    *servus = zeilp + 2;        /* Begruessungszeile                            */
    zeilp += 6;
    *konjfirst = zeilp;         /* Descriptor der ersten Konjugation merken */
    while(**zeilp != '$')       /* Uebrige Konjugationen ueberspringen          */
        zeilp += 2;
    zeilp += 2;                 /* Korrektur                                    */
    keyp = keylist;             /* Adressen der Schluesselgruppen in Feld       */
    i = 0;                      /* 0 = Schluesselwort, 1 = Antwort              */
    while(*zeilp != 0) {        /* bis Dateiende                                */
        *keyp++ = zeilp;        /* Start Schluessel/ Antwortgruppe              */
        if(i)                   /* 1 bedeutet Antwortgruppe                     */
            *keyp++ = zeilp;    /* Antwortgruppe doppelt ablegen (ist           */
        i = !i;                 /* noetig                                       */
```




```
        while(**zeilp != '$')    /* Uebrige Saetze der Gruppe ueberspringen  */
            zeilp += 2;
        zeilp += 2;
    }
    *keyp = 0;                   /* Ende der Schluesselworttabelle           */
}

/*
    Zeile von der Tastatur holen; dabei Rueckgabe der Zeilenlaenge, ent-
    fernung aller Apostrophe "'" aus Text und Umwandlung in Kleinbuchstaben
*/

holeeing(eingabe, einlen)
char eingabe[];
int *einlen;
{
    char *ziel, *quelle, c;

    puts(": ");                 /* Meldung: Zur Eingabe bereit               */
    getns(eingabe + 1, 77);     /* Zeile holen, maximal 77 Zeichen           */
    puts("\n");
    ziel = quelle = eingabe;
    *ziel = ' ';                /* Leerzeichen voranstellen                  */
    while((c = *quelle++) != NULL)  /* Zeichen aus Puffer bis Ende erreicht */
        if(c != '\'')    /* Wenn kein "'" wieder ablegen, aber ohne Luecken  */
            (c < 'A' || c > 'Z') ? *ziel++ = c : *ziel++ = c + 32;
    *ziel++ = ' ';              /* dabei Umwandlung in Kleinbuchstaben       */
    *ziel++ = ' ';              /* 2 Leerzeichen anfuegen                    */
    *ziel = NULL;               /* Endemarkierung                            */
    *einlen = ziel - eingabe;   /* Laenge berechnen                          */
}

/*
    Standard-Antwortsatz anhand der Eingabe bestimmen: Durchsuchen der
    Eingabe nach Schluesselwoertern; wenn gefunden, waehlen eines beliebigen
    Antwortsatzes aus der zugehoerigen Gruppe
*/

findantw(antwort, antlen, restpos, eingabe, einlen, keylist)
char *antwort[], eingabe[];
int keylist[], *antlen, *restpos, einlen;
{
    char **zeilp;
    int *keyp, pos;

    for(keyp = keylist; *keyp != 0; keyp += 3)    /* Schleife Wort-gruppe    */
        for(zeilp = *keyp; **zeilp != '$'; zeilp += 2) /* next Schluessel    */
            if(pos = instr(1, eingabe, einlen, *zeilp, *(zeilp + 1))) {
                *restpos = pos + *(zeilp + 1); /* Gefunden: Rueckgabe Pos.  */
                zeilp = *(keyp + 2);           /* hinter dem Schluesselwort */
                *antwort = *zeilp++;           /* Rueckgabe Adresse Antwort */
                *antlen = *zeilp++;            /* Rueckgabe Antwortlaenge   */
                if(**zeilp != '$') {           /* Beim naechstenmal naechste*/
                    *(keyp + 2) = zeilp;       /* passende Antwort          */
                } else {
                    *(keyp + 2) = *(keyp + 1); /* nach letzter passender    */
                }                              /* wieder bei erster beginnen*/
                return;
            }
}

/*
    Nebensatz konjugieren: In Tabelle konj stehen Paarweise Descriptoren auf
    gegenseitig auszutauschende Worte. z.B. 'your' in 'my' und umgekehrt.
*/

konjugation(neben, nlen, konj)
char neben[];
int *nlen, *konj;
{
    char temp[MAXLINE], **zeilp, *ziel, *quelle, c;
    int l, ss, ls, sr, lr, s, r, len;

    len = *nlen;                /* Anfangslaenge des Nebensatzes             */
    strcpy(temp, neben);        /* wird Schrittweise durch Ergebnsi ersetzt  */
    for(zeilp = konj; **zeilp != '$'; zeilp += 4) {  /* '$'ist Tabellendende*/
        l = 1;                  /* Startposition zum suchen                  */
        ss = *zeilp;            /* Descriptoren eines Wortpaares holen       */
        ls = *(zeilp + 1);
        sr = *(zeilp + 2);      /* Adresse und Laenge des 2. Wortes          */
        lr = *(zeilp + 3);
        while( (s=instr(l,neben,len,ss,ls)) + (r=instr(l,neben,len,sr,lr))) {
            if(s < r ? s : !r) {     /* Test ob 2. Wort in 1. oder umgekehrt */
                strcpy(temp + s - 1, sr);  /* erstes Wort in temp einfuegen */
                strcpy(temp + s - 1 + lr, neben + s - 1 + ls);
                l = s + lr - 1;          /* Rueckuebersetzung sperren        */
                len = len + lr - ls;     /* neue Laenge der Konjugation      */
            } else {    /* Wenn 2. Wort zuerst gefunden, in 1. umwandeln     */
                strcpy(temp + r - 1, ss);
                strcpy(temp + r - 1 + ls, neben + r - 1 + lr);
                l = r + ls - 1:
                len = len + ls -lr;
            }
            strcpy(neben, temp);     /* altuelle Version des strings holen  */
        }
    }
    ziel = neben;            /* doppelte Leerzeichen am Zeilenanfang entfernen */
    quelle = *(neben + 1) != ' ' ? neben : neben + 1;
    while((c = *quelle++) != NULL)                     /* bis zum Zeilenende  */
```

Listing 1. Smalltalk mit Eliza (Fortsetzung)

Records in Pascal »structs« aus je zwei Integerzahlen bestehen. Diese Möglichkeit bieten allerdings nicht alle Compiler.

Die Funktion »instr(start,text,tlen, such,slen)« entspricht der gleichnamigen Basic-Funktion. Ab der Position »start« (gezählt wird hier ab 1) wird in der Zeichenkette »text« nach dem ersten Auftreten der Zeichenkette »such« gesucht. Wenn »such« gefunden wurde, wird die Position zurückgegeben, an der sie in »text« stand. Andernfalls erhält das Hauptprogramm den Wert 0. Die Suche beginnt nicht unbedingt am Anfang von »text«, sondern an der Position, die »start« angibt. Zur Geschwindigkeitssteigerung werden der Funktion mit »tlen« und »slen« auch noch die Längen von Text und Suchstring übergeben. Wenn diese Werte unbekannt sind, kann im Aufruf immer noch »strlen(text)« und »strlen(such)« statt »tlen« und »slen« verwendet werden.

Da die Routine sehr oft durchlaufen wird, spart man hier nicht an der Programmlänge, sondern lieber an der Anzahl der insgesamt auszuführenden Operationen. Wenn die Suchzeichenkette die Länge 1 hat, wird eine besonders einfache Schleife durchlaufen, die nur eine Abfrage auf »Zeichen gefunden« enthält. Ist die Suchzeichenkette länger als ein Zeichen, wird in der äußeren Schleife zunächst nur geprüft, ob das erste Zeichen der Suchzeichenkette mit einem Zeichen des Textes übereinstimmt. Erst wenn eine Übereinstimmung auftaucht, wird die Schleife aufgebaut, die feststellt, ob auch noch die folgenden Zeichen passen. Sobald ein Unterschied festgestellt wird, bricht die Schleife ab. Eine weitere Beschleunigung ist möglich, wenn Ihr Compiler Registervariablen bietet. Sogar »static«-Variablen sind noch wesentlich schneller greifbar als »automatic«-Variablen.

Die Funktion »holeeing(eingabe, &einlen)« holt eine Eingabezeile von der Tastatur. In der Integer-Variablen »einlen« wird die Anzahl der eingegebenen Zeichen zurückgemeldet. Die Routine gibt vor der Eingabe selbständig ein Eingabeprompt (analog zum »A>« in CP/M) aus, und beendet die Eingabe auf dem Bildschirm durch einen Zeilenvorschub. Vor der Rückgabe werden alle eingegebenen Zeichen in Kleinbuchstaben umgewandelt und Apostrophe »'« entfernt. Vor der Zeile stehen jeweils ein und dahinter zwei Leerzeichen.

Die Funktion »konjugiere(&antwort, &anlen,konj)« ist die komplizierteste Funktion des Programms und eine spezielle Eliza-Funktion. Die vollständige Verarbeitung eines Konjugationswort-

paares entspricht einem Durchlauf der for-Schleife. Zunächst werden die Längen und die Adressen der beiden Wörter geholt. Die while-Schleife wird so oft passiert, bis feststeht, daß weder das eine noch das andere Wort sich im Nebensatz befindet. Wenn mindestens eines der beiden gefunden wird, wird das bearbeitet, das im Nebensatz zuerst auftaucht. Der temporäre String enthält zunächst eine Kopie des Nebensatzes. Der Ersatzstring wird an die Stelle des temporären Strings geschrieben, an der das zu ersetzende Wort steht und der Rest des Nebensatzes, der erhalten bleibt, angehängt. Danach wird die neue Länge des Nebensatzes berechnet und die Startposition zum Suchen so weitergesetzt, daß das eben ausgetauschte Wort nicht noch einmal untersucht wird. Zum Schluß wird der temporäre String wieder auf den Nebensatz kopiert. Nach der vollständigen Konjugation werden eventuelle doppelte Leerzeichen vor und hinter dem Nebensatz sowie alle Unterstreichungszeichen entfernt. Zurückgegeben wird die neue Länge des Nebensatzes.

Jedem seine Eliza

Die Funktion »findantw« gibt zu einer Eingabe den passenden Standard-Antwortsatz und die Position des Nebensatzes in der Eingabe zurück. Dazu durchsucht die Funktion alle Schlüsselwortgruppen nach einer Übereinstimmung in der Eingabezeile. Für jede Schlüsselwortgruppe sind im Feld »konj« drei Einträge reserviert. Der erste ist ein Zeiger auf die Descriptoren der Schlüsselwörter, der zweite ein Zeiger auf den ersten Descriptor der dazu passenden Antworten und der dritte ein Zeiger auf die aktuelle Antwort. Sobald ein Schlüsselwort gefunden wurde, liefert die Funktion die Descriptorinhalte der aktuellen Antwort. Danach richtet sich der Zeiger auf den descriptor der nächsten Antwort. Wenn diese Antwort nicht mehr zur Gruppe gehört (Inhalt »$«!), wird wieder die erste Antwort zur aktuellen. So ist sichergestellt, daß alle zum Schlüsselwort passenden Antworten irgendwann an der Reihe sind.

Wenn Sie nun weiter interessiert sind, können Sie darangehen, das Programm auszubauen. Vielleicht statten Sie es mit einem Gedächtnis aus, so daß auch, wenn gerade kein Schlüsselwort in der Eingabe vorkommt, Eliza feststellen kann, welches Hauptthema in der Luft liegt. Oder Sie testen, wie sich eine andere Suchreihenfolge (etwa nach der Lage im Text statt nach der Reihenfolge in der Datei) auf die Antworten auswirkt.

(Helmut Tischer/hg)

```
        if(c != '_')         /* alle Unterstriche entfernen (falls bei bestimm-*/
            *ziel++ = c;     /* ten Worten Markierung fuer bereits ersetzt     */
    *ziel = NULL;            /* notwendig war                                  */
    *nlen = ziel - neben;    /* neue Zeilenlaenge zurueckgeben                 */
}

/*
    Verlassen der Eingabeschleife durch 'bye'. Ein Stern '*' in einem Antwort-
    statz veranlasst das Einfuegen eines konjugierten Nebensatzes. Der Neben-
    satz ist der Teil der Eingabe, hinter dem Schluesselwort
*/

main(argc, argv)
int argc;
char *argv[];
{
    int descript[MAXZEILEN], *keyp, keylist[MAXKEYS],
    einlen, anlen, restpos, t, npos, nlen;
    char datei[DATEILAENGE], eagabe[MAXLINE], oldeing[MAXLINE],neben[MAXLINE],
      *name, *antwort, **servus, **konj, **zeilp;

    if(argc <= 1) {                          /* Datei einlesen und ordnen       */
        name = "ELIZA.DAT";                  /* wenn keine Dateinamensangabe:   */
    } else {                                 /* Standard ist 'ELIZA.DAT'        */
        name = argv[1];
    }
    einlesen(name, datei, descript);
    ordnen(descript, &servus, &konj, keylist);
    for(zeilp = descript; **zeilp != '$'; zeilp += 2) {
        puts(*zeilp);                        /* Headerzeilen anzeigen           */
        puts("\n");
    }
    puts(*servus);                           /* Begruesung von Eliza            */
    puts("\n");
    *oldeing = NULL;                         /* alte Eingabe zunaechst loeschen */
    for(;;) {                                /* Hauptschleife des Programms     */
        do {  /* solange abfragen, bis Eingabe unterschiedlich zur letzten     */
            holeeing(eagabe, &einlen);
            if(!strcmp(eagabe, " bye  "))    /* Programmabbruch                 */
                exit();
            if(t = !strcmp(eagabe,oldeing))  /* Eingabe war schon mal da        */
                puts("please don't repeat yourself !\n");
        } while(t);
        strcpy(oldeing, eagabe);             /* neue Eingabe wird alte          */
        findantw(&antwort, &anlen, &restpos, eagabe, einlen, keylist);
        if(npos = instr(1, antwort, anlen, "*", 1)) { /*Nebensatz einfuegen?*/
            strcpy(neben, " "); /* Nebensatz beginnt hinter Schluesselwort */
            strcpy(neben + 1, eagabe + restpos - 1);
            nlen = einlen - restpos + 2;            /* Laenge des Nebensatzes */
            konjugation(neben, &nlen, konj);        /* Konjugieren            */
            strcpy(eagabe, antwort);    /* Standard-Antwort nicht zerstoeren!*/
            strcpy(eagabe + npos - 1, neben);/* Nebensatz in Kopie einfuegen*/
            strcpy(eagabe + npos - 1 + nlen, antwort + npos);
            antwort = eagabe;               /* Kopie wird zu neuer Antwort    */
        }
        puts(antwort);              /* Gesamtantwort auf Bildschirm ausgeben  */
        puts("\n");
    }
}
/* Geschafft ! */
```

Listing 1. Smalltalk mit Eliza (Schluß)

```
 /
 /
 /
            E L I Z A
            =========
 /
                - der Computer als Psychoanalytiker
 /
Wortschatz: entnommen aus Creative Computing, Juli-August 1977
Programm:   neu entwickelt im M(rz 1986 von Isar-Amper-Soft
 /
 /
     Erz(hl' doch Eliza einfach irgendetwas,
                        wozu Du gerade Lust hast.
 /
 /
$
hi, i am eliza. tell me your problem.
$
 are / am /
were /was /
 you / i /
your /my /
 ive / youve /
 im / youre /
$
can you
$
don't you believe that i can*
perhaps you would like to be able to*
you want me to be able to*
$
can i
$
perhaps you don't want to*
do you want to be able to*
$
you are
youre
$
```

Listing 2. Elizas Antworten als Datei

```
what makes you think i am*
does it please you to believe i am*
perhaps you would like to be*
do you sometimes wish you where*
$
i dont
$
don't you really*
why don't you*
do you wish to be able to*
does that trouble you?
$
i feel
$
tell me more about such feelings.
do you often feel*
do you enjoy feeling*
$
why dont you
$
do you really believe i don't*
perhaps in good time i will*
do you want me to*
$
why cant i
$
do you think you should be able to*
why can't you*
$
are you
$
why are you interestet in whether or not i am*
whould you prefer if i where not*
perhaps in your fantasies i am*
$
i cant
$
how do you know you can't*
have you tried?
perhaps you can now*
$
i am
im /
$
did you come to me because you are*
how long have you been*
do you believe it is normal to be*
do you enjoy being*
$
you /
$
we were discussing you-- not me.
oh, i*
you're not really talking about me, are you?
$
i want
$
what would it mean to you if you got*
why do you want*
suppose you soon got*
what if you never got*
i sometimes also want*
$
what
how
who
where
when
why
$
why do you ask?
does that question interest you?
what answer would please you the most?
what do you think?
are such questions in your mind often?
what is that you really want to know?
have you asked anyone else?
have you asked such questions before?
what else comes to mind when you ask that?
$
name
$
names don't interest me.
i don't care about names-- please go on.
$
cause
$
is that the real reason?
don't any other reasons come to mind?
doeas that reason explain anything else?
what other reasons might there be?
$
sorry
$
please don't apologize!
apologies are not necessary.
what feelings do you have, when you apologize?
don't be so defensive!
$
dream
$
what does that dream suggest to you?
do you dream often?
what persons appear in your dreams?
are aou disturbed by your dreams?
$
hello
hi /
$
how do you do ... please state your problem.
$
maybe
$
you don't seem quite certain.
why the uncertain tone?
can't you be more positive?
you aren't sure?
don't you know?
$
 no
$
why no*
don't say no. its always so negative.
why not?
are you sure?
why no?
$
your
$
why are you concerned about my*
what about your own*
$
always
$
can't you think of a specific example?
when?
what are you thinking of?
really, always?
$
think
$
do you really think so?
but you are not sure you*
do you doubt you*
in what way?
$
alike
$
what resemblance do you see?
what does the similarity suggest to you?
what other connections do you see?
could there really be some connections?
how?
you seem quite positive.
are you sure?
$
yes
$
i see.
i understand.
$
friend
$
why do you bring up the topic of friends?
do your friends worry you?
do your friends pick on you?
are you sure you have any friends?
do you impose on your friends?
perhaps your love for friends worries you.
$
computer
$
do computers worry you?
are you talkink about me in partikular?
are you frightened by machines?
why do you mention computers?
what do you think machines have to do with your problems?
don't you think computers can help people?
what is it about machines that worries you?
$
money
$
why do you have problems with money?
do you think money is everything?
are you sure that money is the problem?
$
eliza
$
i think we want to talk about you, not about me.
whats about me?
why do you always bring up my name?
$
 /
$
say, do you have any psychological problems?
what does that suggest to you?
i see.
i'm not sure i understand you fully.
come elucidate your thoughts.
can you elaborate on that?
that is quite interesting.
$
```

Listing 2. Elizas Antworten als Datei (Schluß)